Linköping Studies in Science and Technology

Dissertation No. 937

TALplanner and Other Extensions to Temporal Action Logic

by

Jonas Kvarnström

Linköpings universitet
INSTITUTE OF TECHNOLOGY

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2005

Parts of this doctoral thesis appear in other publications:

Doherty, P., & Kvarnström, J. (2001). TALplanner: A temporal logic-based planner. *AI Magazine*, *22*(3), 95–102. See also http://www.aaai.org/Library/Magazine/Vol22/22-03/vol22-03.html.

Doherty, P., & Kvarnström, J. (1999). TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. In Dixon, C., & Fisher, M. (Eds.), *Proceedings of the Sixth International Workshop on Temporal Representation and Reasoning (TIME'99)*, pp. 47–54, Orlando, Florida, USA. IEEE Computer Society Press. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/time99-final.ps.gz.

Doherty, P., & Kvarnström, J. (1998). Tackling the qualification problem using fluent dependency constraints: Preliminary report. In Khatib, L., & Morris, R. (Eds.), *Proceedings of the Fifth International Workshop on Temporal Representation and Reasoning (TIME-98)*, pp. 97–104, Los Alamitos, California, USA. IEEE Computer Society Press.

Gustafsson, J., & Kvarnström, J. (2001). Elaboration tolerance through object-orientation. In *Proceedings of the Fifth Symposium on Logical Formalizations of Commonsense Reasoning (Common Sense-2001)*. Available at http://www.cs.nyu.edu/faculty/davise/commonsense01/final/kvarnstrom.ps.

Gustafsson, J., & Kvarnström, J. (2004). Elaboration tolerance through object-orientation. *Artificial Intelligence*, *153*, 239–285. © 2003 Elsevier B. V.

Kvarnström, J. (2002). Applying domain analysis techniques for domain-dependent control in TALplanner. In Ghallab, M., Hertzberg, J., & Traverso, P. (Eds.), *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, pp. 101–110, Toulouse, France. AAAI Press, Menlo Park, California, USA.

Kvarnström, J., & Doherty, P. (2000a). Tackling the qualification problem using fluent dependency constraints. *Computational Intelligence*, *16*(2), 169–209. © 2000 Blackwell Publishers, 350 Main Street, Malden, MA 02148, USA, and 108 Cowley Road, Oxford, OX4 1JF, UK.

Kvarnström, J., & Doherty, P. (2000b). TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, *30*, 119–169.

Kvarnström, J., Doherty, P., & Haslum, P. (2000). Extending TALplanner with concurrency and resources. In Horn, W. (Ed.), *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-2000)*, Frontiers in Artificial Intelligence and Applications, pp. 501–505, Berlin, Germany. IOS Press, The Netherlands. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/www-ecai.ps.gz.

Kvarnström, J., & Magnusson, M. (2003). TALplanner in the Third International Planning Competition: Extensions and control rules. *Journal of Artificial Intelligence Research*, *20*, 343–377. Available at http://www.jair.org/contents/v20.html.

Abstract

Though the exact definition of the boundary between intelligent and non-intelligent artifacts has been a subject of much debate, one aspect of intelligence that many would deem essential is *deliberation*: Rather than reacting "instinctively" to its environment, an intelligent system should also be capable of *reasoning* about it, reasoning about the effects of actions performed by itself and others, and creating and executing plans, that is, determining which actions to perform in order to achieve certain goals. True deliberation is a complex topic, requiring support from several different sub-fields of artificial intelligence. The work presented in this thesis spans two of these partially overlapping fields, beginning with *reasoning about action and change* and eventually moving over towards *planning*.

The qualification problem relates to the difficulties inherent in providing, for each action available to an agent, an exhaustive list of all qualifications to the action, that is, all the conditions that may prevent this action from being executed in the intended manner. The first contribution of this thesis is a framework for *modeling qualifications* in Temporal Action Logic (TAL).

As research on reasoning about action and change proceeds, increasingly complex and interconnected domains are modeled in increasingly greater detail. Unless the resulting models are structured consistently and coherently, they will be prohibitively difficult to maintain. The second contribution of this thesis is a framework for *structuring TAL domains* using object-oriented concepts.

Finally, the second half of the thesis is dedicated to the task of *planning*. TLPlan pioneered the idea of using domain-specific control knowledge in a temporal logic to constrain the search space of a forward-chaining planner. We develop a new planner called TALplanner, based on the same idea but with several new extensions, some of which are enabled by fundamental differences in the way the planner verifies that a plan satisfies control formulas. TALplanner generates concurrent plans and can take resource constraints into account. The planner also applies several new automated domain analysis techniques to control formulas, further increasing performance by orders of magnitude for many problem domains.

Acknowledgements

Although I had always planned to leave the university after my undergraduate studies, some of the last courses I took served to pique my interest in research and made me consider applying to become a graduate student. One of them was a course in Knowledge Representation, given by Patrick Doherty, who later became my supervisor and main thesis advisor. I would like to thank him both for (unknowingly) helping to lead me towards this path and for his help and support over the years.

I am also grateful to all of my colleagues in the AIICS division for a great deal of help and inspiration. I would especially like to thank Joakim Gustafsson, with whom I have co-authored one of the first articles in this thesis. Working in the same room during the first part of my graduate studies led to many interesting discussions, both strictly research-related and otherwise. At a later stage, Patrik Haslum and Martin Magnusson co-authored two articles on the subject of planning. Once again, this lead to some very fruitful discussions, without which this thesis would probably have been quite different.

Finally, I would like to thank my mother and father, and my friends at work and outside work, for their support and encouragement during these years. I couldn't have done this without you.

This work has been supported in part by the Wallenberg Foundation, the Swedish Research Council for Engineering Sciences (TFR) and the ECSEL / ENSYM graduate studies program.

Contents

I Introduction and Background **1**

1 Introduction **3**
- 1.1 Background: Reasoning about Action and Change . . . 3
- 1.2 TAL and the Qualification Problem . . . 4
- 1.3 Growing Pains: Modeling Complex Domains . . . 5
- 1.4 A New Task: Moving towards Planning . . . 6
- 1.5 Brief Contents . . . 8

2 TAL: Temporal Action Logics **9**
- 2.1 History . . . 9
- 2.2 Basic Concepts . . . 13
- 2.3 The TAL-C Surface Language $\mathcal{L}$(ND) . . . 14
- 2.4 The TAL-C Base Language $\mathcal{L}$(FL) . . . 22

II Extensions to TAL **29**

3 Tackling the Qualification Problem using Fluent Dependency Constraints **31**
- 3.1 Abstract . . . 31
- 3.2 Introduction . . . 32
- 3.3 The Qualification Problem . . . 34
- 3.4 The Russian Airplane Hijack Scenario . . . 42
- 3.5 TAL-Q: Temporal Action Logic with Qualification . . . 43
- 3.6 Representing the RAH Scenario . . . 43
- 3.7 Representing the Qualification Problem in TAL-Q . . . 46
- 3.8 Additional Aspects of the Qualification Problem . . . 50
- 3.9 Alternative Approaches to the Qualification Problem . . . 54
- 3.10 Additional Examples . . . 56
- 3.11 Comparisons . . . 59
- 3.12 Conclusion . . . 65

4 Elaboration Tolerance through Object-Orientation 71
4.1 Introduction . . . 71
4.2 The TAL family and the TAL-C Logic . . . 73
4.3 Basic Object-Oriented Modeling in TAL-C . . . 73
4.4 Inheritance and Overriding . . . 79
4.5 Additional Object-Oriented Concepts . . . 81
4.6 Elaboration Tolerance through Object-Orientation . . . 83
4.7 Missionaries and Cannibals . . . 84
4.8 Elaborations of the MCP Domain . . . 90
4.9 Solving the Missionaries and Cannibals Problems . . . 102
4.10 Traffic World . . . 109
4.11 Related Work . . . 109
4.12 Conclusions . . . 110
4.13 Acknowledgements . . . 110

III TALplanner 111

5 Planning 113
5.1 Introduction to Planning . . . 115
5.2 Forward-Chaining Planning . . . 128
5.3 Blockhead, TLPlan, and Control Formulas . . . 132

6 TALplanner 139
6.1 An Overview of TALplanner . . . 139
6.2 Representing Planning Problems in TAL . . . 142
6.3 The Basic TALplanner Algorithm . . . 163
6.4 TALplanner with TAL-based Control Rules . . . 167
6.5 Tense Control Rules and Progression . . . 179
6.6 Completeness, Control and the Definition of Plans . . . 184
6.7 Evaluation vs Progression: Initial Benchmark Tests . . . 186

7 Concurrency and Resources 191
7.1 Concurrent TALplanner . . . 192
7.2 Preventing Interference in Concurrent Plans . . . 195
7.3 Modeling Limited Resources . . . 201
7.4 Concurrency, Resources and Cycle Checking . . . 207
7.5 Concurrent TALplanner . . . 208

8 Domain Analysis Techniques for Domain-Dependent Control 211
8.1 General Optimization Framework . . . 215
8.2 Equivalence Optimizations . . . 216
8.3 Context-Dependent Optimizations . . . 217
8.4 Using State Invariants . . . 220

8.5 Eliminating Quantifiers 222
8.6 Generating Precondition Control 230
8.7 Empirical Benchmark Tests 231
8.8 Related Work 237

9 Planning Competitions **239**
9.1 International Planning Competition 2000 240
9.2 International Planning Competition 2002 246

10 Discussion **281**
10.1 TALplanner and the WITAS Project 281
10.2 Hand-Tailored versus Fully Automated Planning 282
10.3 Using Control Rules 284
10.4 Using TAL in TALplanner 285
10.5 Future Work 287
10.6 Acknowledgments 288

Bibliography **289**

Part I

Introduction and Background

Chapter 1

Introduction

Today, there is an abundance of "intelligent" consumer electronics available in any electronics store. My new digital camera has "artificial intelligence auto-focus", determining automatically how to adjust the focus settings so that the most relevant parts of a picture are in focus. My twelve-year-old stereo system has an "AI" button, which determines the best order in which to copy tracks from a CD to a tape, and believe it or not, my laptop has an "intelligent battery pack".

But ever since its inception in the 1950s, the field of artificial intelligence (AI) has always aimed much higher than this. Though the exact definition of the boundary between intelligent and non-intelligent artifacts has been a subject of much debate, one aspect of intelligence that many would deem essential is *deliberation*: An intelligent system should not only be able to react "instinctively" to its environment, but should also be capable of *reasoning* about it, reasoning about the effects of actions performed by itself and others within this environment, and creating and executing plans, that is, determining which actions to perform in order to achieve certain goals. Sadly, this ability appears to be missing from my laptop battery.

True deliberation is a complex topic, requiring support from several different sub-fields of AI. The work presented in this thesis spans two of these partially overlapping fields, beginning with *reasoning about action and change* (RAC) and eventually moving over towards *planning*.

1.1 Background: Reasoning about Action and Change

The field of reasoning about action and change (RAC) is concerned with reasoning about dynamic worlds, where properties of the world change over time due to actions being invoked by the reasoner (and possibly others) as well as due to processes taking place in the world. The tasks a reasoner might be expected to perform in such a domain include *prediction*, determining what will happen if certain actions are performed, and *postdiction*, inferring facts about the world at earlier timepoints given some knowledge of what did happen as a result of certain actions being taken. *Planning*, determining which actions to perform in order to achieve a given goal, would also seem to fit into the boundaries of reasoning about action and change but is nevertheless generally considered to be a separate research area.

Performing these reasoning tasks naturally requires some information about the world, and this information should preferably be represented in some principled and structured manner, in a form amenable to automated reasoning. Thus, any formalism for *reasoning* about action and change is generally developed concurrently with a formalism for *modeling* a dynamic domain and the actions that can be performed therein, and there is considerable overlap between this field and the area of *knowledge representation* (KR).

A large number of different approaches to modeling and reasoning about actions have been proposed in the literature, some of them applicable only to very limited domains and some closer to being suitable for real world problems.

Many researchers, ever since McCarthy (1959) wrote about programs with common sense, have used logic as the main means for representing knowledge as well as for actually reasoning about the facts that are known to be true and the actions that could be performed to change these facts. Two of the main advantages of this approach are that logic provides a succinct means for describing incomplete knowledge about one's environment, which is certainly necessary in any real-world scenario, and that there is a well-defined formal semantics, which is a prerequisite for being able to trust the conclusions that can be drawn from a model of a domain.

As for any other approach to knowledge representation and reasoning, logic-based approaches naturally have their own set of difficulties that need to be dealt with and overcome – otherwise, KR and RAC would no longer be active research areas. Some of these difficulties are representational in nature, related to the problem of finding compact and comprehensible ways of describing certain kinds of knowledge. This tends to lead to extending the power of a logic, for example by going from first-order to second-order logic in order to be able to use techniques such as circumscription (McCarthy, 1980). Other problems are computational, since inference in a first-order or higher-order logic is not necessarily the most efficient way of performing a task, which often leads to attempts to find more restricted logics that are nevertheless powerful enough for the task at hand. Fortunately there has been steady progress towards solving many of these problems, and the applicability and efficiency of the logics and reasoning mechanisms being used is continuously being improved.

1.2 TAL and the Qualification Problem

Some of the more powerful logics currently used in the area of reasoning about action and change belong to the Temporal Action Logics (TAL) family. These logics have been developed specifically for reasoning about action and change, and therefore provide explicit representations for time, in order to reason about dynamic aspects of a domain, as well as fluents (state variables whose values change over time), actions, and other entities.

TAL originates in a logic called PMON (Doherty, 1994), which was considerably less complex than current TAL logics but nevertheless provided an unusually robust and flexible solution to the *frame problem* (McCarthy & Hayes, 1969) – essentially, how to succinctly specify all the facts about the world that are *not* changed by each action. Extensions to PMON lead to the logic PMON-RC (Gustafsson & Doherty, 1996), which provided one solution to the representational aspect of the *ramification problem* (named by Finger, 1987) – the problem that any action can have many side effects, some of which may trigger other side effects in finite or infinite chains, and there should be a modular way of describing such indirect effects rather than specifying them in a monolithic action definition.

Further extensions included the logic PMON^{+} (Doherty, 1996), later renamed TAL 1.0, and the logic TAL-C (Chapter 2; Karlsson & Gustafsson, 1999), which introduced support for concurrent actions. But even with the extensions made in these logics, only two of the three standard problems in reasoning about action and change using logic had been tackled within the PMON/TAL framework: The *qualification problem* had still been left unexplored.

The qualification problem relates to the fact that the set of preconditions for an action is usually far larger than we would intuitively think. Some of these conditions would appear to be quite natural, such as the need to have the right car key in order to start a car, and are best modeled as preconditions within the main action specification. Some are less obvious, such as the fact that there should not be a potato in the tailpipe, that no-one should have put sugar in the gasoline and that the engine should not have been removed. If such conditions need to be modeled explicitly, they should be specified as separate constraints rather than as part of a monolithic action specification, in order to improve modularity.

Solving this aspect of the qualification problem in the context of TAL was the focus of an article coauthored with Patrick Doherty (Doherty & Kvarnström, 1998; Kvarnström & Doherty, 2000a). This article forms the first topic of this thesis (Chapter 3).

1.3 Growing Pains: Modeling Complex Domains

At this stage in the evolution of TAL, it was possible to model far more complex domains than in the original PMON logic that had been used a few years earlier. Instead of tiny sequential toy scenarios that could be formalized in three short logic formulas, we had now modeled a number of medium-sized domains with indirect effects, qualified actions, and concurrency, formalized in dozens of TAL formulas. This had worked out very well, and it was time to find new, even more complex domains to be modeled in order to truly test the applicability and scalability of the TAL logics.

Though this was a very promising development, it was quickly becoming clear that the old ways of structuring domains according to statement type – all action definitions in one section, all indirect effects in another section, and so on – were no longer sufficient when domains grew in size and complexity. It was all too easy to end up with the logical equivalent of "spaghetti code", making it difficult to verify the correctness of the domain descriptions and to reuse suitable parts from one domain when modeling new but similar domains. In order to avoid this, it was necessary to find a more principled modeling strategy.

Gustafsson started working on this problem and decided to apply object-oriented modeling techniques to TAL. After some time I also got involved in the effort, and our work resulted in a paper (Gustafsson & Kvarnström, 2001) describing how applying object-orientation to reasoning about action and change could lead to more structured models exhibiting a higher degree of elaboration tolerance (McCarthy, 1998). An extended version of this paper was accepted for publication in Artificial Intelligence (Gustafsson & Kvarnström, 2004), and forms the second topic of this thesis.

1.4 A New Task: Moving towards Planning

Even before the object-oriented extensions were developed, the TAL logics had grown quite powerful. But despite this, they could in themselves only be used for reasoning about what *would* happen given a fixed set of actions to be performed, or what *had* happened given a fixed set of actions that had been performed. These reasoning tasks are important, but so is the ability to determine which actions *should* be performed in order to achieve a certain goal within a given environment – in other words, the ability to create *plans*.

Our first concrete application for planning was found at the end of 1998 in the WITAS project (Doherty, Haslum, Heintz, Merz, Persson, & Wingman, 2004; Doherty, Granlund, Kuchcinski, Sandewall, Nordberg, Skarman, & Wiklund, 2000; Doherty, 2004; Heintz & Doherty, 2004b; Merz, 2004), a research project aimed at developing an autonomous UAV (an unmanned aerial vehicle, in this case a helicopter). This UAV would be able to fly and navigate autonomously, and would be able to perform a number of different tasks using its onboard computer and various sensors mounted on the platform, including delivering packages, tracking and following cars and other vehicles, detecting conventional and unconventional (possibly dangerous) maneuvers performed by vehicles, and assisting emergency vehicles in finding the best path to their destination.

The software running on board the UAV would include image recognition systems, geographical information systems, and newly developed helicopter control software handling all the intricacies of flying a helicopter in varying environments.

It would also include numerous high-level deliberative systems, supported by a path planner (Pettersson, 2003; Pettersson & Doherty, 2004) finding paths around obstacles and most likely also an action planner generating action sequences for various tasks to be performed by the helicopter. Unfortunately none of the planners available at the time appeared to be completely suitable for this project. Many planners were unable to handle operators with complex preconditions and conditional effects. Most could not generate true concurrent plans, where the execution intervals of two operators can overlap completely or partially, which would be required in order to take proper advantage of the parallelism inherent in the domain. Despite a focus on efficiency, and despite significant progress being made in the field, most algorithms that were available at the time did not appear to be efficient enough to be run in real time on board an autonomous helicopter, requiring minutes to solve even rather small problem instances from common benchmark domains. And finally, despite the fact that the TAL logics were developed specifically for reasoning about action and change, making them eminently suitable for use in a planner, there were no planners based on the TAL semantics for actions. This would have facilitated interfacing the planner to other parts of the higher level deliberative systems on board the UAV, where it was envisioned that TAL would be used as a modeling language for the UAV domain.

As a first step towards developing a planner for the UAV domain, we wanted to take an existing planning algorithm and adapt it to the use of TAL, creating a new implementation that supported a limited subset of the TAL syntax and semantics and then integrating this implementation with the existing tools for TAL. This prototype would be limited to sequential plans and would not necessarily be very efficient, but it would nevertheless provide a way of learning more about the state of the art in planning, about the difficulties specific to integrating a planner with TAL, and about the requirements a planner would need to satisfy in order to be useful in the project. After this prototype was finished, it would be time to consider whether the initial approach was suitable for our needs or whether another planning paradigm would perhaps be more amenable to being used in the project.

One planning algorithm which seemed quite interesting was TLPlan (Bacchus & Kabanza, 1996b, 2000), which used a forward-chaining search technique with a twist: A set of temporal logic formulas could be used to constrain the plans that were generated, allowing users to guide the planner towards the goal in a way that could increase planning efficiency by several orders of magnitude for a number of common benchmark domains. This was not the first planner to allow users to specify control information, but it did so in a flexible and rather intuitive manner, and the use of a temporal logic for control would be quite compatible with the modeling of actions and planning domains in a temporal action logic such as TAL.

It seemed reasonable to believe that much if not all of the efficiency gains that were possible through the additional pruning would carry over to the UAV planning domain. These efficiency gains would be extremely significant given the nature of the UAV project, and would be well worth the additional effort that would have to be spent when creating suitable domain-specific control rules, making the TLPlan approach a prime candidate for use in the project. Another advantage of this approach was that it was reasonably expressive – and although TLPlan itself did not support all the expressivity that would be required for the UAV project, its forward-chaining nature meant that it would most likely be feasible to develop a variation of the planner that would support concurrency, metric time and other desirable extensions while staying within the basic framework of forward-chaining with control rules. Consequently, we decided to use this framework as the basis for a new planner: TALplanner.

Adapting the idea of using control rules to TAL was not too difficult, and since it was possible to reuse some code from VITAL (Kvarnström, 2005), a tool for visualizing and reasoning about domain models formalized in TAL, the first phase of the project took only a couple of months. To our surprise, the new prototype TALplanner was also considerably faster than TLPlan, though given that the high-level algorithms were very similar in this version, this would mainly have to be due to low-level algorithms (for state storage and formula evaluation) and differences in implementation.

After this initial prototype was evaluated, it was decided that the TALplanner project should continue, and from this point in time TALplanner has developed independently from TLPlan. A number of extensions have been developed, including both increases in expressivity, support for concurrency and resources, and efficiency improvements through optimizations as well as new formula analysis techniques developed specifically for TALplanner. Most of the extensions have been published in a number of papers and articles (Doherty & Kvarnström, 1999; Kvarnström & Doherty, 2000b; Kvarnström, Doherty, & Haslum, 2000; Doherty & Kvarnström, 2001; Kvarnström, 2002; Kvarnström & Magnusson, 2003), and these form the basis for a new, updated and integrated description of TALplanner in the final part of this thesis.

1.5 Brief Contents

The first part of this thesis mainly provides some background for my work. It consists of the introduction you have just read together with a chapter describing the PMON and TAL logics and their evolution over the years (Chapter 2).

The second part of the thesis contains two chapters related to the use of TAL in reasoning about action and change. Chapter 3 contains a journal article on tackling the qualification problem in TAL, while Chapter 4 contains an article on achieving elaboration tolerance through object-oriented modeling in TAL.

Finally, the third part of the thesis (Chapters 5 through 10) describes the current version of TALplanner.

Chapter 2

TAL: Temporal Action Logics

Logic is often used as the main means for reasoning about actions and the way they affect the environment in which the reasoner is operating. This chapter introduces Temporal Action Logics (TAL), one of several frameworks used for reasoning about action and change in logic. Though no new results will be presented in this chapter, understanding TAL will be important when reading other parts of this thesis.

2.1 History

TAL is not a single logic, but a family of logics. Some of these logics are extensions to older versions, while others branch out to provide alternative approaches to solving particular types of reasoning problems, often being incorporated into the main branch of the TAL development tree after having proven their usefulness.

The logic which will be used in much of this thesis is called TAL-C. Before defining TAL-C, though, we will take a look at the history that led to the development of the TAL logics.

2.1.1 The Beginning

In the beginning, there was chaos.

To be more exact (and somewhat less dramatic), quite a few logics for reasoning about action and change have been proposed over the years, but for a long time there was no principled formal method for determining whether a logic always yielded the intended results.

Given a (possibly partial) description of a world and the actions taking place in that world, one would like to have some kind of formal guarantee that the conclusions that can be drawn from that description, the facts that can be inferred from that logical theory, are sound and preferably also complete. Proving this might be reasonably straight-forward for an ordinary monotonic first order logic, where there is more or less universal agreement on what soundness and completeness means, but a logic for reasoning about action and change is usually expected to be non-monotonic. While adding new facts to a theory in a monotonic logic always extends the set of valid conclusions that should be drawn from this theory, non-monotonic logics are able to "jump to conclusions", and adding new facts may invalidate some previous conclusions. This leads to the question of exactly what conclusions should be jumped to, and when they should be retracted.

For example, non-monotonic inferences are used in many solutions to the *frame problem* (McCarthy & Hayes, 1969), the problem that when you formally define what happens when you perform an action (such as moving a box from A to B), you only want to specify the facts that will be changed (the box will now be at location B) and not all the myriad facts that will *not* be changed (the box will not change color, other boxes will not move, the speed of the car going by outside will not be affected, and so on). A logic for reasoning about action and change should automatically infer, or jump to the conclusion, that the only actions that occur are those that are explicitly specified and that all facts that are not explicitly stated to change at any given point in time will indeed remain unchanged – but if a new action is added to the theory, some of these conclusions will have to be withdrawn. For example, if an action is added that paints a block red, it should no longer be possible to conclude that no blocks ever change colors. This non-monotonicity, where previous conclusions are no longer valid when new facts are added, renders the classical definition of soundness for first-order logic useless.

Without a suitable formal framework providing a definition of what *should* be entailed by a theory, logics could only be tested against a researcher's intuitive idea of how they should work. This was usually done with the help of a small set of benchmark examples for which the logic did yield the intended conclusions. Unfortunately logic sometimes does not behave intuitively, and now and then new problem instances were found for which some existing logics gave some rather surprising results. For example, some logics had a partial solution to the frame problem that initially appeared correct but would allow you to infer that if you load a gun, wait, and shoot, the gun might magically become unloaded during the waiting action (for some reason these benchmark tests are often quite violent). This scenario is also known as the Yale Shooting Problem (Hanks & McDermott, 1986).

Once such a deficiency was found in a logic, the logic might have to be discarded. Even if the logic could be patched to handle the new test, the presence of one flaw indicated that there could still be others waiting to be found, making it difficult to completely trust any of the logics to provide the intended results.

2.1.2 Features and Fluents

In his book *Features and Fluents* (1994), Sandewall developed a formal framework for assessing the correctness (soundness and completeness) of a logic for reasoning about action and change.

The events taking place in the Yale Shooting Problem could be seen as an *action scenario*, a kind of story line or plot that tells us what has happened or what will happen. Like other approaches to reasoning about action and change using logic, the Features and Fluents framework provides a way of formally describing such scenarios. An *action scenario description* contains a set of observations of facts that hold at various points in time (the gun is not loaded in the initial state at time 0), together with a generic definition of actions that can be performed (the Fire action means that if the gun is loaded when the action is invoked, the turkey will be dead at the end of the action) and a specification of which actions do in fact occur (Fire occurs between time 5 and time 7). Action scenario descriptions are sometimes called *scenario descriptions* or *narratives*.

Reasoning problems were then classified according to their ontological and epistemological characteristics.

The *ontological characteristics* relates to the structure of the (abstract) world an agent is reasoning about, and specifies properties such as whether inertia can be assumed (that is, whether facts only change when explicitly caused to change), whether surprises can occur, whether actions can be performed concurrently, and whether effects of actions can be delayed. Note that explicitly specifying these characteristics often provides additional information about the world in which the scenario is taking place, information that cannot necessarily be inferred from the action scenario description itself.

The *epistemological characteristics* specifies constraints on the knowledge provided to the agent by the world, such as whether the agent knows all actions that are performed, whether all preconditions for being able to execute a given action are explicitly known to the agent, whether it knows all effects of all actions, and whether it has complete knowledge about the initial state.

Sandewall then defined the exact conclusions that a logic should be able to produce from a reasoning problem belonging to a number of such classes of characteristics. A set of preferential entailment methods[1] were developed, many of which corresponded directly to the behavior of existing logics for reasoning about action and change. These preferential entailment methods were then analyzed, giving upper and lower bounds in terms of the classes of reasoning problems for which they produced exactly the intended conclusions.

PMON, Pointwise Minimization of Occlusion with Nochange premises, was one of the few preferential entailment methods that were assessed correct for the $\mathcal{K}$–**IA** class of action scenario descriptions, where $\mathcal{K}$ is an epistemological characteristics stating approximately that explicit, correct and accurate knowledge is provided (with no requirements on complete knowledge in the initial state and no restrictions on knowledge about other states), and **IA** is an ontological character-

[1]Preferential entailment reduces the set of classical models of a theory by only retaining those models that are minimal according to a given preference relation, a strict partial order over logical interpretations (Shoham, 1987).

istics stating approximately that discrete integer time is used together with plain inertia (without surprises or other complicating factors). Though ramifications and qualifications were not allowed in $\mathcal{K}$–**IA**, the class is in fact quite broad, permitting the use of conditional effects, non-deterministic effects, and incomplete specification of states and the timing of actions.

2.1.3 PMON and TAL

While the original PMON was a preferential entailment method, Doherty later developed an equivalent classical logic, with a circumscription axiom capturing the PMON definition of preferential entailment (Doherty, 1994; Doherty & Łukaszewicz, 1994). This new logic is also called **PMON**, and uses two languages for representing and reasoning about narratives. The surface language $\mathcal{L}(\mathrm{SD})$, Language for Scenario Descriptions, provides a convenient high-level notation for describing narratives, and can be seen as a set of macros easily translated into the base language $\mathcal{L}(\mathrm{FL})$, which was initially a many-sorted first-order language and was later altered to be an order-sorted[2] first-order language. The $\mathcal{L}(\mathrm{SD})$ language was later renamed to $\mathcal{L}(\mathrm{ND})$, Language for Narrative Descriptions.

Though the circumscription axiom was a second-order formula, PMON action definitions were structured in a way that guaranteed the possibility of using quantifier elimination techniques to reduce the axiom to a first-order formula, enabling the use of standard first-order theorem proving techniques to reason about PMON narratives.

The original PMON logic was further extended and generalized in several steps, while still retaining the use of the original base $\mathcal{L}(\mathrm{FL})$ together with the possibility to reduce circumscribed narratives to first-order logic. Although the extended logics belong to what we now call the TAL family, each is essentially an incremental addition to the base logic PMON.

PMON-RC, proposed by Gustafsson and Doherty (1996), provides a solution to the ramification problem for a broad, but as yet unassessed class of action scenarios. The main idea is the addition of a new statement type for causal constraints, where changes taking place in the world can automatically trigger new changes at the same timepoint or at a specified delay from the original change. The solution is very fine-grained in the sense that one can easily encode dependencies between individual objects in the domain, work with both boolean and non-boolean fluents and represent both Markovian and non-Markovian dependencies (Giunchiglia & Lifschitz, 1995). PMON-RC also correctly handles chains of side effects.

PMON^{+}, developed by Doherty (1996), is an extended version of the original PMON logic incorporating the changes made in PMON-RC together with additional extensions. This logic was later renamed **TAL 1.0**.

[2]Essentially, an order-sorted language allows the use of sub-sorts; for example, `car` and `bicycle` may be sub-sorts of the `vehicle` sort.

TAL-C, proposed by Karlsson and Gustafsson (1999), uses fluent dependency constraints (an extended form of causal constraints) as a basis for representing concurrent actions. A number of phenomena related to action concurrency such as interference between one action's effects and another's execution, bounds on concurrency, and conflicting, synergistic, and cumulative effects of concurrent actions are supported.

TAL-C has been used as the basis for several articles in this thesis. This logic will be described below in sufficient detail for understanding those articles, omitting some details regarding type structures and constraints on sorts in order to improve readability. We refer the reader to Karlsson and Gustafsson (1999) and Doherty, Gustafsson, Karlsson, and Kvarnström (1998) for a complete definition of this logic and to Doherty (1994), Doherty and Łukaszewicz (1994) and Gustafsson and Doherty (1996) for further background information and descriptions of earlier logics in the TAL family.

2.2 Basic Concepts

We assume there is an *agent* interested in reasoning about a specific *world*. This world might be formally defined, or it might be the "real world", in which case the agent can only reason about a formally defined abstraction of the real world. In either case, it is assumed that the world is dynamic, in the sense that the various properties or *features* of the world can change over time.

The formal version of the Yale Shooting Problem could be seen as an abstraction of a real shooting situation, where the only properties being modeled are loaded and alive, two simple boolean features representing whether the gun is loaded and whether the turkey is alive, respectively.

The TAL framework also permits the use of multiple *value domains*, which can be used for modeling different types of *objects* that might occur in the world which is being modeled. For example, the well-known blocks world contains blocks that can be stacked on top of each other. The blocks world can be modeled using a value domain for blocks, containing values such as **A**, **B** and **C**, together with features such as $\mathsf{on}(block_1, block_2)$, which holds iff $block_1$ is on top of $block_2$, and $\mathsf{clear}(block)$, which holds iff there is no block on top of the given block. Of course, values can also be used to represent properties of objects rather than the objects themselves. For example, if the color of each block should be modeled, then this could be done using a value domain for colors containing values such as **red**, **green** and **blue**, together with a color-valued (non-boolean) feature $\mathsf{color}(block)$.

Time itself could be viewed differently depending on the nature of the world being reasoned about and the reasoning abilities of the agent. TAL uses linear time, as opposed to branching time, and allows the use of either continuous real-valued time or discrete integer time. Research within the TAL framework has mostly been

		time 0	time 1	time 2	time 3	time 4	
	on(A,A)	false	false	false	false	false	
	on(A,B)	true	false	false	false	false	
	on(B,A)	false	false	false	false	true	**fluent**
	on(B,B)	false	false	false	false	false	
	ontable(A)	false	false	true	true	true	
	ontable(B)	true	true	true	false	false	
	clear(A)	true	false	true	true	false	
feature	clear(B)	false	true	true	false	true	
	handempty	true	false	true	false	true	
				state			

Figure 2.1: Viewing a Development as Fluents or States

focused on discrete non-negative integer time, and this will be used throughout this thesis.

The development of the world over a (possibly infinite) period of discrete time can be viewed in two different ways. Figure 2.1 shows what would happen in a simple blocks world scenario where block **A** is initially on top of **B**, which is on the table, and where one unstacks **A** from **B**, places it on the table, picks up **B**, and finally stacks this block on top of **A**. The information about this scenario can be viewed as a sequence of *states*, where each state provides a value to all features (or "state variables") for a single common timepoint, or as a set of *fluents*, where each fluent is a function of time which specifies the development of a single feature. We sometimes use the terms "feature" and "fluent" interchangeably to refer to either a specific property of the world or the function specifying its value over time.

Since there is an agent, there is usually also a set of *actions* that the agent can perform. Such actions can only be performed when the requisite *preconditions* are satisfied. Performing an action changes the state of the world according to a set of given rules. Such rules are not necessarily deterministic. For example, the action of tossing a coin can be modeled within the TAL framework, and there will be two possible result states.

All of these concepts are modeled in the language $\mathcal{L}$(ND). We will now provide an overview of this language and the translation from $\mathcal{L}$(ND) to the order-sorted first-order base language $\mathcal{L}$(FL).

2.3 The TAL-C Surface Language $\mathcal{L}$(ND)

A narrative in $\mathcal{L}$(ND) can be said to consist of two parts: The *narrative background specification* (NBS), which provides background information that is common to all narratives for a particular domain, and the *narrative specification* (NS), which provides information specific to a particular instance of a reasoning problem.

All information about a narrative is represented as a set of labeled narrative statements in $\mathcal{L}(\text{ND})$, except for the vocabulary, which defines the constant symbols, feature symbols, action symbols, and other symbols that are available for use in narrative formulas. Since narrative examples used in the literature have traditionally been quite simple, the vocabulary has usually either been considered to be implicit in the remainder of the narrative specification or been described informally in the main text of the article. In this thesis, however, vocabularies will generally be described in terms of labeled narrative declaration statements using a syntax borrowed from the software tools VITAL (Kvarnström, 2005) and TALplanner (Chapter 6).

Before providing a formal definition of the $\mathcal{L}(\text{ND})$ language, we will introduce most of the macros, formula types and statement classes using an example narrative: An extended version of the Hiding Turkey Scenario (Sandewall, 1994).

Example 2.3.1 (Extended Hiding Turkey Scenario)
In the extended hiding turkey scenario, there is a turkey that may or may not be deaf, and there is also a gun. First, we load the gun. Loading the gun makes some noise, and unless the turkey is deaf, it will hide whenever there is noise. However, if the turkey has been hiding for a while and there has been no noise, the turkey will decide to come out in the open again. After a while, we fire the gun, and if the turkey is not hiding at that time, it will die.

This scenario can be represented in TAL-C as the following narrative, the components of which will be described in further detail in the following sections:

domain	turkey :elements { **T1** }
domain	gun :elements { **G1** }
feature	alive(turkey), deaf(turkey), hiding(turkey) :domain boolean
feature	loaded(gun) :domain boolean
feature	noise :domain boolean
action	Load(gun), Fire(gun,turkey)

per1 $\forall t, turkey\ [Per(t, \mathsf{alive}(turkey)) \wedge Per(t, \mathsf{deaf}(turkey)) \wedge Per(t, \mathsf{hiding}(turkey))]$

per2 $\forall t, gun\ [Per(t, \mathsf{loaded}(gun))]$

per3 $\forall t\ [Dur(t, \mathsf{noise}, \mathbf{false})]$

dom1 $\forall t, turkey\ [[t]\ \mathsf{noise} \wedge \neg\mathsf{deaf}(turkey) \rightarrow [t+1]\ \mathsf{hiding}(turkey)]$

acs1 $[t_1, t_2]\ \mathsf{Load}(gun) \rightsquigarrow R((t_1, t_2]\ \mathsf{loaded}(gun)) \wedge I((t_1, t_2]\ \mathsf{noise})$

acs2 $[t_1, t_2]\ \mathsf{Fire}(gun, turkey) \rightsquigarrow$
$([t_1]\ \mathsf{loaded}(gun) \wedge \neg\mathsf{hiding}(turkey) \rightarrow R((t_1, t_2]\ \neg\mathsf{alive}(turkey))) \wedge$
$([t_1]\ \mathsf{loaded}(gun) \rightarrow R((t_1, t_2]\ \neg\mathsf{loaded}(gun)))$

dep1 $\forall t, turkey\ [[t]\ \neg\mathsf{hiding}(turkey) \wedge \neg\mathsf{deaf}(turkey) \wedge C_T([t]\ \mathsf{noise}) \rightarrow$
$R([t+1]\ \mathsf{hiding}(turkey))]$

dep2 $\forall t, turkey\ [[t, t+9]\ \mathsf{hiding}(turkey) \wedge \neg\mathsf{noise} \rightarrow R([t+10]\ \neg\mathsf{hiding}(turkey))]$

obs1	[0] alive(**T1**) ∧ ¬loaded(**G1**) ∧ ¬hiding(**T1**)
occ1	[1,4] Load(**G1**)
occ2	[5,6] Fire(**G1**,**T1**)

Since the narrative does not specify whether or not the turkey was deaf, there will be two classes of models; one where the turkey is deaf, does not hide, and ends up being shot, and one where it hears the noise, hides, and emerges from hiding ten timepoints later. ■

2.3.1 Narrative Background Specification

In the narrative background specification, *persistence statements* (labeled **per**) allow each fluent to be specified as being persistent (normally retaining its value from the previous timepoint), durational (normally reverting to a default value), or dynamic (varying freely, subject to other constraints involving this fluent). *Domain constraints* (labeled **dom**) characterize acausal information which is always true in the world being modeled. *Action type specifications* (labeled **acs**) provide generic definitions of action types, while *dependency constraints* (labeled **dep**) characterize causal dependencies among fluents. The narrative background specification also contains the vocabulary for the narrative.

Vocabulary

The vocabulary for the hiding turkey scenario requires three value domains: The standard boolean domain, together with two domains for turkeys and guns. There are five boolean fluents (alive, deaf, hiding, loaded, and noise), some of which take turkeys or guns as parameters, and there are two actions (Load and Fire).

domain	turkey :elements { **T1** }
domain	gun :elements { **G1** }
feature	alive(turkey), deaf(turkey), hiding(turkey) :domain boolean
feature	loaded(gun) :domain boolean
feature	noise :domain boolean
action	Load(gun), Fire(gun,turkey)

The boolean domain is always present in all narratives, and behaves as if it had been specified in the following manner:

domain	boolean :elements { **true**, **false** }

Persistence Statements

Intuitively, the first four fluents in the extended hiding turkey scenario describe properties that do not change unless something changes them, while the fifth, noise, is different – there is no noise unless someone is currently making noise. This distinction between persistent and durational fluents is important. A persistent fluent can only change when an action or dependency constraint allows it to change (the *persistence assumption* or *inertia assumption*). Otherwise, it retains the same value it had at the previous timepoint. A durational fluent is associated with a default value, and can only take on another value when an action or dependency constraint allows it to (the *default value assumption*). At timepoints when no action or dependency constraint explicitly allows it to take on another value, it will immediately revert to its default value. Though this is not used in this narrative, a fluent can also be *dynamic* if it is not declared to be persistent or durational. Since no persistence or default value assumption is applied, dynamic fluents can vary freely over time to satisfy observations and domain constraints.

Whether a fluent is persistent or durational – or neither – is defined in a set of *persistence statements*, using the $\mathcal{L}(\text{ND})$ macros *Per* and *Dur*. For a persistent fluent f, $Per(t, f)$ should be true, and for a durational fluent f with default value ω, $Dur(t, f, \omega)$ should be true; the temporal argument allows persistence properties to vary over time, though this flexibility is usually not used. Note that some earlier TAL logics used a fixed *nochange axiom* instead of persistence statements, forcing all fluents to be persistent. Using persistence statements provides a more flexible and fine-grained approach to controlling the default behavior of fluents.

per1 $\forall t, \mathit{turkey}\ [Per(t, \mathsf{alive}(\mathit{turkey})) \land Per(t, \mathsf{deaf}(\mathit{turkey})) \land Per(t, \mathsf{hiding}(\mathit{turkey}))]$
per2 $\forall t, \mathit{gun}\ [Per(t, \mathsf{loaded}(\mathit{gun}))]$
per3 $\forall t\ [Dur(t, \mathsf{noise}, \mathbf{false})]$

Domain Constraints

Domain constraints represent knowledge about logical fluent dependencies which are not specific to a particular reasoning problem instance but which are known to hold in every possible scenario taking place within a domain. In domain constraints, as well as other TAL formulas, the fact that a fluent f takes on a particular value ω is denoted by the elementary fluent formula $f \hat{=} \omega$. For the boolean domain, the formula $f \hat{=} \textbf{true}$ ($f \hat{=} \textbf{false}$) can be abbreviated f ($\neg f$). Elementary fluent formulas can be combined using boolean connectives and quantification over values to form fluent formulas. The fixed fluent formula $[\tau]\ \phi$ states that the fluent formula ϕ holds at the timepoint τ.

Although no domain constraints are strictly needed for this scenario, we will show one possible constraint as an example: If there is noise at some timepoint, then all turkeys that are not deaf will be hiding at the next timepoint.

dom1 $\forall t, \mathit{turkey}\ [[t]\ \mathsf{noise} \land \neg\mathsf{deaf}(\mathit{turkey}) \rightarrow [t+1]\ \mathsf{hiding}(\mathit{turkey})]$

Action Types

Actions can be invoked by the agent in order to change some properties in the world. Loading a gun g in the extended hiding turkey scenario should cause

$\mathsf{loaded}(g)$ to become true, for example. But since the loaded fluent is persistent, simply stating that $\mathsf{loaded}(g)$ will be true at the end of the action invocation is not sufficient. Instead, it is necessary to use a *reassignment macro* to explicitly release this fluent from the persistence assumption at the specific point in time where it should change values from false to true.

There are three different reassignment macros: X, R and I. They can all be used with a temporal interval, for example $R((\tau, \tau'] \ \alpha)$, or a single timepoint, for example $I([\tau] \ \alpha)$. Each of these operators has the effect of releasing the fluents occurring in α from the persistence and default value assumptions during the given interval or at the given timepoint. However, the operators differ in whether they place further constraints on the values of these fluents, and if so, at what time.

The X operator is used for *occlusion*. Its purpose is simply to allow the value of the fluents in the formula α to vary at a timepoint or during an interval, and therefore it does not further constrain the fluents occurring in α. Intuitively, the X operator occludes (hides) any changes in a fluent value from the persistence or default value constraints generated by the persistence statements in the narrative.

The R operator is used for *reassignment*, and ensures that α will hold at the final timepoint in the interval. During the rest of the interval, the fluents occurring in α are allowed to vary freely, unaffected by the persistence or default value assumption (but still subject to other constraints that may also be present in the narrative).

The I operator is used for *interval reassignment* and is often but not always used in combination with durational fluents. It ensures that α will hold during the entire interval.

An *action type specification* uses reassignment macros to define what will happen if and when a particular action is invoked. Note that it does not state that an action does occur; this is specified in the narrative specification using action occurrence statements.

In the extended hiding turkey scenario, there are two actions at our disposal. We can Load a gun (**acs1**), which ensures that this particular gun is loaded when the action has been executed but also makes some noise throughout the duration of the action: The action definition forces noise to be true in the entire interval $(t_1, t_2]$, and thereafter noise will automatically revert to its default value, **false**. We can also Fire a gun (**acs2**), which results in the gun no longer being loaded – and if the gun was loaded when the Fire action was invoked, and the turkey we were aiming at was not hiding, the turkey will no longer be alive.

acs1 $[t_1, t_2] \ \mathsf{Load}(gun) \ \rightsquigarrow \ R((t_1, t_2] \ \mathsf{loaded}(gun)) \ \wedge \ I((t_1, t_2] \ \mathsf{noise})$

acs2 $[t_1, t_2] \ \mathsf{Fire}(gun, turkey) \ \rightsquigarrow$
$([t_1] \ \mathsf{loaded}(gun) \ \wedge \ \neg\mathsf{hiding}(turkey) \ \rightarrow \ R((t_1, t_2] \ \neg\mathsf{alive}(turkey))) \ \wedge$
$([t_1] \ \mathsf{loaded}(gun) \ \rightarrow \ R((t_1, t_2] \ \neg\mathsf{loaded}(gun)))$

Dependency Constraints

Actions must be explicitly triggered using action occurrence statements. Some changes or activities in the world are instead triggered by conditions that hold or that become true in the world. Such changes can be modeled using *dependency constraints*.

An interesting property of the turkeys in this domain is that they are afraid of sounds: If a turkey is not deaf and there is some noise, it will immediately hide. This fact cannot be modeled as an acausal domain constraint, since such constraints cannot provide a sufficient cause for the noise fluent to change values. Neither can it be modeled by an action, since actions must be invoked explicitly and cannot be triggered automatically by conditions that hold in the world. Instead, it is modeled using a dependency constraint. The constraint **dep1** states that if a turkey is not deaf and not hiding, and a noise *begins* (the C_T macro, "changes to true"), then at the next timepoint, hiding is explicitly assigned the value **true**. Similarly, if a turkey has been hiding for ten timepoints, and there has been no noise during that time, it will stop hiding (**dep2**).

dep1 $\forall t, turkey\ [[t]\ \neg\mathsf{hiding}(turkey)\ \wedge\ \neg\mathsf{deaf}(turkey)\ \wedge\ C_T([t]\ \mathsf{noise})\ \rightarrow$
$R([t+1]\ \mathsf{hiding}(turkey))]$

dep2 $\forall t, turkey\ [[t, t+9]\ \mathsf{hiding}(turkey)\ \wedge\ \neg\mathsf{noise}\ \rightarrow\ R([t+10]\ \neg\mathsf{hiding}(turkey))]$

2.3.2 Narrative Specification

In the narrative specification, *observation statements* (labeled **obs**) are intended to represent observations of fluent values at specific timepoints while *action occurrence statements* (labeled **occ**) specify which instances of the generic action types occur and during which time intervals.

Observation Statements

Observation statements are intended to describe specific facts that have been observed to hold in the world. They provide information about a particular reasoning problem instance within a domain, and are therefore part of the narrative specification. In the example scenario, a number of facts have been observed in the initial state: The turkey is alive but not hiding, and the gun is not loaded. We do not observe whether the turkey is deaf or not, and there is no need to state that there is no noise, since the durational fluent noise is false by default.

obs1 $[0]\ \mathsf{alive}(\mathbf{T1})\ \wedge\ \neg\mathsf{loaded}(\mathbf{G1})\ \wedge\ \neg\mathsf{hiding}(\mathbf{T1})$

Action Occurrence Statements

Action occurrence statements specify which actions actually do take place in a narrative. Like observations, they are part of the narrative specification – the instance-specific part of the narrative.

For the hiding turkey scenario, two action occurrence statements are required, specifying that the gun is loaded and fired.

occ1	$[1,4]$ $\mathsf{Load}(\mathbf{G1})$
occ2	$[5,6]$ $\mathsf{Fire}(\mathbf{G1},\mathbf{T1})$

2.3.3 The Language $\mathcal{L}(\mathrm{ND})$

This section defines the surface language $\mathcal{L}(\mathrm{ND})$. The translation to the first-order language $\mathcal{L}(\mathrm{FL})$ is presented in Section 2.4.1. The overline is used as abbreviation of a sequence, when the contents of the sequence are obvious. For example, $f(\overline{x},\overline{y})$ means $f(x_1,...,x_n,y_1,...,y_m)$.

Definition 2.3.1 (Basic Sorts)
There are a number of sorts for values $\mathcal{V}_i$, including the boolean sort $\mathcal{B}$ with the constants $\{\mathbf{true},\mathbf{false}\}$. TAL is order-sorted, and a sort may be specified to be a subsort of another sort. The sort $\mathcal{V}$ is a supersort of all value sorts.

There are a number of sorts for features $\mathcal{F}_i$, each one associated with a value sort $dom(\mathcal{F}_i) = \mathcal{V}_j$ for some j. The sort $\mathcal{F}$ is a supersort of all fluent sorts.

There is also a sort for actions $\mathcal{A}$ and a temporal sort $\mathcal{T}$. ■

The sort $\mathcal{T}$ is assumed to be interpreted, but can be axiomatized in first-order logic as a subset of Presburger arithmetics (Koubarakis, 1994) (natural numbers with addition).

Definition 2.3.2 (Terms)
A *value term*, often denoted by ω, is a variable v or a constant $\mathbf{v}$ of sort $\mathcal{V}_i$ for some i, an expression $value(\tau,f)$ where τ is a temporal term and f is a fluent term, or an expression $\mathsf{g}(\omega_1,\ldots,\omega_n)$ where $\mathsf{g} : \mathcal{V}_{k_1} \times \ldots \times \mathcal{V}_{k_n} \rightarrow \mathcal{V}_i$ is a value function symbol and each ω_j is a value term of sort $\mathcal{V}_{k_j}$.

A *temporal term*, often denoted by τ, is a variable t or a constant $0,1,2,3,\ldots$ or $\mathsf{s}_1,\mathsf{t}_1,\ldots$, or an expression of the form $\tau_1 + \tau_2$, all of sort $\mathcal{T}$.

A *fluent term*, often denoted by f, is a feature variable or a feature expression $\mathsf{f}(\omega_1,\ldots,\omega_n)$ where $\mathsf{f} : \mathcal{V}_{k_1} \times \ldots \times \mathcal{V}_{k_n} \rightarrow \mathcal{F}_i$ is a feature symbol and each ω_j is a value term of sort $\mathcal{V}_{k_j}$.

An *action term* Ψ is an expression $A(\omega_1,\ldots,\omega_n)$ where $A : \mathcal{V}_{k_1} \times \ldots \times \mathcal{V}_{k_n} \rightarrow \mathcal{A}$ is an action symbol and each ω_j is a value term of sort $\mathcal{V}_{k_j}$. ■

Variables are typed and range over the values belonging to a specific sort. Although the sort is sometimes specified explicitly in narratives, it is more common to simply give the variable the same name as the sort but (like all variables) written in italics, possibly with a prime and/or an index. For example, the variables *turkey*, *turkey*$'$ and *turkey*$_3$ would be of the sort `turkey`. Similarly, variables named t or τ are normally temporal variables, and variables named n are normally integer-valued variables.

The function $\mathit{value}(\tau, f)$ returns the value of the fluent f at the timepoint τ, where $[\tau]\ f \hat{=} v$ iff $\mathit{value}(\tau, f) = v$. The expression $[\tau]\ f \hat{=} g$, where f and g are fluent terms, is shorthand notation for $[\tau]\ f \hat{=} \mathit{value}(\tau, g)$.

Definition 2.3.3 (Temporal and Value Formulas)
If τ and τ' are temporal terms, then $\tau = \tau'$, $\tau < \tau'$ and $\tau \leq \tau'$ are *temporal formulas*. A *value formula* is of the form $\omega = \omega'$ where ω and ω' are value terms, or $\mathsf{r}(\omega_1, \ldots, \omega_n)$ where $\mathsf{r} : \mathcal{V}_{k_1} \times \ldots \times \mathcal{V}_{k_n}$ is a relation symbol and each ω_j is a value term of sort $\mathcal{V}_{k_j}$. ■

We will sometimes write $\tau \leq \tau' < \tau''$ to denote the conjunction $\tau \leq \tau' \wedge \tau' < \tau''$, and similarly for other combinations of the relation symbols $\leq$ and $<$.

Definition 2.3.4 (Fluent Formula)
An *elementary fluent formula*, sometimes called an *isvalue expression*, has the form $f \hat{=} \omega$ where f is a fluent term of sort $\mathcal{F}_i$ and ω is a value term of sort $\mathit{dom}(\mathcal{F}_i)$. A *fluent formula* is an elementary fluent formula or a combination of fluent formulas formed with the standard logical connectives and quantification over values. ■

The elementary fluent formula $f \hat{=} \mathbf{true}$ ($f \hat{=} \mathbf{false}$) can be abbreviated f ($\neg f$).

Definition 2.3.5 (Timed Formulas)
Let τ and τ' be temporal terms and α a fluent formula. Then:

- $[\tau, \tau']\ \alpha$, $(\tau, \tau']\ \alpha$, $[\tau, \tau')\ \alpha$, $(\tau, \tau')\ \alpha$, $[\tau, \infty)\ \alpha$, $(\tau, \infty)\ \alpha$ and $[\tau]\ \alpha$ are *fixed fluent formulas*,
- $C_T([\tau]\ \alpha)$, $C_F([\tau]\ \alpha)$ and $C([\tau]\ \alpha)$ are *change formulas*,
- $R([\tau, \tau']\ \alpha)$, $R((\tau, \tau']\ \alpha)$, $R([\tau, \tau')\ \alpha)$, $R((\tau, \tau'])\ \alpha)$ and $R([\tau]\ \alpha)$ are *reassignment formulas*, and
- $X([\tau, \tau']\ \alpha)$, $X((\tau, \tau']\ \alpha)$, $X([\tau, \tau')\ \alpha)$, $X((\tau, \tau'])\ \alpha)$ and $X([\tau]\ \alpha)$ are *occlusion formulas*.

Fixed fluent formulas, change formulas, reassignment formulas and occlusion formulas are called *timed formulas*. ■

Definition 2.3.6 (Static Formula)
A *static formula* is a temporal formula, a value formula, a fixed fluent formula, a change formula, `true`, `false`, or a combination of static formulas formed using the standard logical connectives together with quantification over values and time. ■

Note that the formulas `true` and `false` are not the same as the boolean values **true** and **false**.

Definition 2.3.7 (Change Formula)
A *change formula* is a formula that has (or is rewritable to) the form $Q\overline{v}(\alpha_1 \vee ... \vee \alpha_n)$ where $Q\overline{v}$ is a sequence of quantifiers with variables, and each α_i is a conjunction of static, occlusion and reassignment formulas. The change formula is called *balanced* iff the following two conditions hold. (a) Whenever a feature $f(\overline{\omega})$ appears inside a reassignment or occlusion formula in one of the α_i disjuncts, it must also appear in all other α_i's inside a reassignment or occlusion formula with exactly the same temporal argument. (b) Any existentially quantified variable v in the formula, whenever appearing inside a reassignment or occlusion formula, only does so in the position $f \mathrel{\hat{=}} v$. ■

Definition 2.3.8 (Application Formula)
An *application formula* is any of the following: (a) a balanced change formula; (b) $\Lambda \rightarrow \Delta$, where Λ is a static formula and Δ is a balanced change formula; or (c) a combination of elements of types (a) and (b) formed with conjunction and universal quantification over values and time. ■

Definition 2.3.9 (Occurrence Formula)
An *occurrence formula* has the form $[\tau, \tau']\ \Psi$, where τ and τ' are temporal terms and Ψ is an action term. ■

Definition 2.3.10 (Persistence Formula)
A *persistence formula* is an expression of the form $Per(\tau, f)$ where τ is a temporal term and f is a fluent term, an expression of the form $Dur(\tau, f, \omega)$ where τ is a temporal term, f is a fluent term and ω is a value term, or a combination of persistence formulas formed with conjunction and universal quantification over values or time. ■

Definition 2.3.11 (Narrative Statements)
An action type specification or action schema (labeled **acs**) has the form $[t, t']\ \Psi \rightarrow \phi$, where t and t' are temporal variables, Ψ is an action term and ϕ is an application formula.

A dependency constraint (labeled **dep**) is an application formula.

A domain constraint (labeled **dom**) is a static formula.

A persistence statement (labeled **per**) is a persistence formula.

An observation statement (labeled **obs**) is a static formula.

An action occurrence statement (labeled **occ**) is an occurrence formula $[\tau, \tau']\ \Psi$ where τ and τ' are variable-free temporal terms and Ψ is a variable-free action term. ■

2.4 The TAL-C Base Language $\mathcal{L}(\text{FL})$

In order to reason about a particular narrative, it is first mechanically translated into the base language $\mathcal{L}(\text{FL})$, an order-sorted classical first-order language with

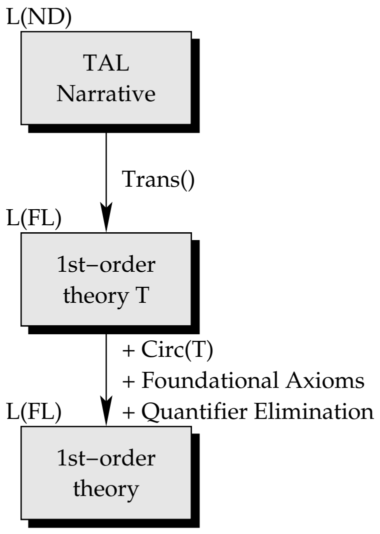


Figure 2.2: The relation between $\mathcal{L}(\text{ND})$ and $\mathcal{L}(\text{FL})$

equality using a linear discrete time structure (Figure 2.2). A circumscription policy is applied to the resulting theory, foundational axioms are added, and quantifier elimination techniques are used to reduce the resulting second order theory to first order logic.

$\mathcal{L}(\text{FL})$ uses the predicates $\mathit{Holds} : \mathcal{T} \times \mathcal{F} \times \mathcal{V}$, $\mathit{Occlude} : \mathcal{T} \times \mathcal{F}$, and $\mathit{Occurs} : \mathcal{T} \times \mathcal{T} \times \mathcal{A}$, where $\mathcal{T}$ is the temporal sort, $\mathcal{F}$ is a supersort of all fluent sorts and $\mathcal{V}$ is a supersort of all value sorts. The *Holds* predicate expresses what value a fluent has at each timepoint, and is used in the translation of fixed fluent formulas; for example, the formula $[0]\ \mathsf{alive}(\mathit{turkey}) \mathrel{\hat{=}} \mathbf{true} \land \mathsf{loaded}(\mathit{gun}) \mathrel{\hat{=}} \mathbf{false}$ can be translated into $\mathit{Holds}(0, \mathsf{alive}(\mathit{turkey}), \mathbf{true}) \land \mathit{Holds}(0, \mathsf{loaded}(\mathit{gun}), \mathbf{false})$. The *Occlude* predicate expresses the fact that a persistent or durational fluent is exempt from its persistence or default value assumption, respectively, at a given timepoint. It is used in the translation of the R, I and X operators, which are intended to change the values of fluents. Finally, the *Occurs* predicate expresses that a certain action occurs during a certain time interval, and is used in the translation of action occurrence statements and action type specifications.

2.4.1 Translation from $\mathcal{L}(\text{ND})$ to $\mathcal{L}(\text{FL})$

The following translation function is used to translate $\mathcal{L}(\text{ND})$ formulas into $\mathcal{L}(\text{FL})$.

Definition 2.4.1 (*Trans* Translation Function)
Trans is called the *expansion transformation*, and is defined as follows. All variables occurring only on the right-hand side are assumed to be fresh variables.

The formulas `true` and `false` need no translation:

$$\begin{aligned} \mathit{Trans}(\texttt{true}) &= \texttt{true} \\ \mathit{Trans}(\texttt{false}) &= \texttt{false} \end{aligned}$$

Basic macros are translated into $\mathcal{L}$(FL) predicates:

$$\begin{aligned} \mathit{Trans}([\tau]\ f(\overline{\omega})) &= \mathit{Holds}(\tau, f(\overline{\omega}), \mathbf{true}) \\ \mathit{Trans}([\tau]\ f(\overline{\omega}) \hat{=} \omega) &= \mathit{Holds}(\tau, f(\overline{\omega}), \omega) \\ \mathit{Trans}(X([\tau]\ f(\overline{\omega}))) &= \mathit{Occlude}(\tau, f(\overline{\omega})) \\ \mathit{Trans}(X([\tau]\ f(\overline{\omega}) \hat{=} \omega)) &= \mathit{Occlude}(\tau, f(\overline{\omega})) \\ \mathit{Trans}([\tau, \tau']\ A(\overline{\omega})) &= \mathit{Occurs}(\tau, \tau', A(\overline{\omega})) \end{aligned}$$

In some versions of TAL, the $\mathcal{L}$(ND) functions *Per* and *Dur* are also translated into $\mathcal{L}$(FL) predicates. Here, they are translated directly into constraints on fluent values and occlusion.

$$\begin{aligned} \mathit{Trans}(\mathit{Per}(\tau, f)) &= \forall t.\tau = t + 1 \wedge \neg \mathit{Occlude}(t+1, f) \rightarrow \\ &\qquad \forall v[\mathit{Holds}(t+1, f, v) \leftrightarrow \mathit{Holds}(t, f, v)] \\ \mathit{Trans}(\mathit{Dur}(\tau, f, \omega)) &= \neg \mathit{Occlude}(\tau, f) \rightarrow \mathit{Holds}(\tau, f, \omega) \end{aligned}$$

Top-level connectives and quantifiers are left unchanged:

$$\begin{aligned} \mathit{Trans}(\neg\alpha) &= \neg \mathit{Trans}(\alpha) \\ \mathit{Trans}(\alpha\ \mathcal{C}\ \beta) &= \mathit{Trans}(\alpha)\ \mathcal{C}\ \mathit{Trans}(\beta), \text{where } \mathcal{C} \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}. \\ \mathit{Trans}(\mathcal{Q}v[\alpha]) &= \mathcal{Q}v[\mathit{Trans}(\alpha)], \text{where } \mathcal{Q} \in \{\forall, \exists\}. \end{aligned}$$

Fixed fluent formulas can contain nested connectives and quantifiers, which are transferred outside the scope of the temporal context $[\tau]$.

$$\begin{aligned} \mathit{Trans}([\tau]\ \mathcal{Q}v[\alpha]) &= \mathcal{Q}v[\mathit{Trans}([\tau]\ \alpha)], \text{where } \mathcal{Q} \in \{\forall, \exists\}. \\ \mathit{Trans}([\tau]\ \neg\alpha) &= \neg \mathit{Trans}([\tau]\ \alpha) \\ \mathit{Trans}([\tau]\ \alpha\ \mathcal{C}\ \beta) &= \mathit{Trans}([\tau]\ \alpha)\ \mathcal{C}\ \mathit{Trans}([\tau]\ \beta), \text{where } \mathcal{C} \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}. \end{aligned}$$

Nested connectives and quantifiers can also occur within occlusion formulas. However, the translation of these formulas has to be modified somewhat to take into account the fact that any occlusion formula should occlude *all* fluents occurring within the scope of the occlusion operator: Even a disjunctive formula such as $X([\tau]\ \alpha \vee \beta)$ should occlude all fluents in α *and* all fluents in β and is therefore not equivalent to $X([\tau]\ \alpha) \vee X([\tau]\ \beta)$ but to $X([\tau]\ \alpha) \wedge X([\tau]\ \beta)$. The translation procedure takes this into account by removing negations inside the X operator, translating connectives occurring inside X into conjunctions, and converting all quantifiers inside X into universal quantification.

$$\begin{aligned} \mathit{Trans}(X([\tau]\ \neg\alpha)) &= \mathit{Trans}(X([\tau]\ \alpha)) \\ \mathit{Trans}(X([\tau]\ \alpha\ \mathcal{C}\ \beta)) &= \mathit{Trans}(X([\tau]\ \alpha) \wedge X([\tau]\ \beta)), \text{where } \mathcal{C} \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}. \\ \mathit{Trans}(X([\tau]\mathcal{Q}v[\alpha])) &= \forall v[\mathit{Trans}(X([\tau]\alpha))], \text{where } \mathcal{Q} \in \{\forall, \exists\}. \end{aligned}$$

Fixed fluent formulas can contain infinite temporal intervals. This is a shorthand notation; infinity is not part of the temporal sort and disappears in the translation.

$$\begin{aligned} \mathit{Trans}([\tau,\infty)\ \alpha) &= \forall t[\tau \leq t \rightarrow \mathit{Trans}([t]\alpha)] \\ \mathit{Trans}((\tau,\infty)\ \alpha) &= \forall t[\tau < t \rightarrow \mathit{Trans}([t]\alpha)] \end{aligned}$$

Finite temporal intervals are permitted both in fixed fluent formulas and in the occlusion operator. Only one form of interval is shown; the extension to allow open, closed and semi-closed intervals is trivial.

$$\begin{aligned} \mathit{Trans}([\tau,\tau']\ \alpha) &= \forall t[\tau \leq t \leq \tau' \rightarrow \mathit{Trans}([t]\alpha)] \\ \mathit{Trans}(X((\tau,\tau']\ \alpha)) &= \forall t[\tau < t \leq \tau' \rightarrow \mathit{Trans}(X([t]\alpha))] \end{aligned}$$

The R and I operators are defined as follows. Again, one form of interval is shown.

$$\begin{aligned} \mathit{Trans}(R((\tau,\tau']\ \alpha)) &= \mathit{Trans}(X((\tau,\tau'],\alpha)) \land \mathit{Trans}([\tau]\alpha) \\ \mathit{Trans}(R([\tau]\ \alpha)) &= \mathit{Trans}(X([\tau]\ \alpha)) \land \mathit{Trans}([\tau]\ \alpha) \\ \mathit{Trans}(I((\tau,\tau']\alpha)) &= \mathit{Trans}(X((\tau,\tau']\ \alpha)) \land \mathit{Trans}((\tau,\tau']\ \alpha) \\ \mathit{Trans}(I([\tau]\ \alpha)) &= \mathit{Trans}(X([\tau]\ \alpha)) \land \mathit{Trans}([\tau]\ \alpha) \end{aligned}$$

Finally, the C_T "changes to true" operator is defined as follows, with the operators C_F (changes to false) and C (changes) added for symmetry.

$$\begin{aligned} \mathit{Trans}(C_T([\tau]\ \alpha)) &= \forall t[\tau = t+1 \rightarrow \mathit{Trans}([t]\neg\alpha)] \land \mathit{Trans}([\tau]\ \alpha) \\ \mathit{Trans}(C_F([\tau]\ \alpha)) &= \forall t[\tau = t+1 \rightarrow \mathit{Trans}([t]\alpha)] \land \mathit{Trans}([\tau]\ \neg\alpha) \\ \mathit{Trans}(C([\tau]\ \alpha)) &= \mathit{Trans}(C_T([\tau]\ \alpha) \lor C_F([\tau]\ \alpha)) \end{aligned}$$ ■

Example 2.4.1 (Extended Hiding Turkey Scenario, continued)
The following is the translation of the Extended Hiding Turkey Scenario into $\mathcal{L}(\mathrm{FL})$. Here, $\neg \mathit{Holds}(\tau, f, \mathbf{true})$ has sometimes been simplified into $\mathit{Holds}(\tau, f, \mathbf{false})$. Free variables are assumed to be universally quantified.

per1 $\forall t, \mathit{turkey}[(\forall t'.t = t'+1 \land \neg \mathit{Occlude}(t'+1, \mathsf{alive}(\mathit{turkey})) \rightarrow$
$\forall \mathit{boolean}.\mathit{Holds}(t'+1, \mathsf{alive}(\mathit{turkey}), \mathit{boolean}) \leftrightarrow \mathit{Holds}(t', \mathsf{alive}(\mathit{turkey}), \mathit{boolean})) \land$
$(\forall t'.t = t'+1 \land \neg \mathit{Occlude}(t'+1, \mathsf{deaf}(\mathit{turkey})) \rightarrow$
$\forall \mathit{boolean}.\mathit{Holds}(t'+1, \mathsf{deaf}(\mathit{turkey}), \mathit{boolean}) \leftrightarrow \mathit{Holds}(t', \mathsf{deaf}(\mathit{turkey}), \mathit{boolean})) \land$
$(\forall t'.t = t'+1 \land \neg \mathit{Occlude}(t'+1, \mathsf{hiding}(\mathit{turkey})) \rightarrow$
$\forall \mathit{boolean}.\mathit{Holds}(t'+1, \mathsf{hiding}(\mathit{turkey}), \mathit{boolean}) \leftrightarrow \mathit{Holds}(t', \mathsf{hiding}(\mathit{turkey}), \mathit{boolean}))]$

per2 $\forall t, \mathit{gun}[(\forall t'.t = t'+1 \land \neg \mathit{Occlude}(t'+1, \mathsf{loaded}(\mathit{gun})) \rightarrow$
$\forall \mathit{boolean}.\mathit{Holds}(t'+1, \mathsf{loaded}(\mathit{gun}), \mathit{boolean}) \leftrightarrow \mathit{Holds}(t', \mathsf{loaded}(\mathit{gun}), \mathit{boolean}))]$

per3 $\forall t[\neg \mathit{Occlude}(t, \mathsf{noise}) \rightarrow \mathit{Holds}(t, \mathsf{noise}, \mathbf{false})]$

dom1 $\mathit{Holds}(t, \mathsf{noise}, \mathbf{true}) \land \neg \mathit{Holds}(t, \mathsf{deaf}(\mathit{turkey}), \mathbf{true}) \rightarrow$
$\mathit{Holds}(t+1, \mathsf{hiding}(\mathit{turkey}), \mathbf{true})$

acs1 $\mathit{Occurs}(t_1, t_2, \mathsf{Load}(\mathit{gun})) \rightarrow$
$\mathit{Holds}(t_2, \mathsf{loaded}(\mathit{gun}), \mathbf{true}) \land$
$\forall t[t_1 < t \leq t_2 \rightarrow \mathit{Occlude}(t, \mathsf{loaded}(\mathit{gun}))] \land$
$\forall t[t_1 < t \leq t_2 \rightarrow \mathit{Holds}(t, \mathsf{noise}, \mathbf{true})] \land$
$\forall t[t_1 < t \leq t_2 \rightarrow \mathit{Occlude}(t, \mathsf{noise})]$

acs2 $\mathit{Occurs}(t_1, t_2, \mathsf{Fire}(\mathit{gun}, \mathit{turkey})) \rightarrow$
$((\mathit{Holds}(t_1, \mathsf{loaded}(\mathit{gun}), \mathbf{true}) \land \mathit{Holds}(t_1, \mathsf{hiding}(\mathit{turkey}), \mathbf{false}) \rightarrow$
$\mathit{Holds}(t_2, \mathsf{alive}(\mathit{turkey}), \mathbf{false}) \land \forall t[t_1 < t \leq t_2 \rightarrow \mathit{Occlude}(t, \mathsf{alive}(\mathit{turkey}))]) \land$
$(\mathit{Holds}(t_1, \mathsf{loaded}(\mathit{gun}), \mathbf{true}) \rightarrow$
$\mathit{Holds}(t_2, \mathsf{loaded}(\mathit{gun}), \mathbf{false}) \land \forall t[t_1 < t \leq t_2 \rightarrow \mathit{Occlude}(t, \mathsf{loaded}(\mathit{gun}))]))$

dep1 $\neg Holds(t, \mathsf{hiding}(turkey), \mathbf{true}) \wedge \neg Holds(t, \mathsf{deaf}(turkey), \mathbf{true}) \wedge Holds(t, \mathsf{noise}, \mathbf{true}) \wedge \forall u[t = u + 1 \rightarrow \neg Holds(u, \mathsf{noise}, \mathbf{true})] \rightarrow Holds(t+1, \mathsf{hiding}(turkey), \mathbf{true}) \wedge Occlude(t+1, \mathsf{hiding}(turkey))$
dep2 $\forall u[t \leq u \wedge u \leq t+9 \rightarrow Holds(u, \mathsf{hiding}(turkey), \mathbf{true}) \wedge \neg Holds(u, \mathsf{noise}, \mathbf{true})] \rightarrow \neg Holds(t+10, \mathsf{hiding}(turkey), \mathbf{true}) \wedge Occlude(t+10, \mathsf{hiding}(turkey))$
obs1 $Holds(0, \mathsf{alive}(\mathbf{T1}), \mathbf{true}) \wedge Holds(0, \mathsf{loaded}(\mathbf{G1}), \mathbf{false}) \wedge Holds(0, \mathsf{hiding}(\mathbf{T1}), \mathbf{false})$
occ1 $Occurs(1, 4, \mathsf{Load}(\mathbf{G1}))$
occ2 $Occurs(5, 6, \mathsf{Fire}(\mathbf{G1}, \mathbf{T1}))$ ■

2.4.2 Circumscription Policy

The logical theory which is the result of the translation is still under-constrained in the sense that a number of implicit assumptions about fluent change in the world remain to be characterized. In general, we want to encode the blanket assumption that fluent values do not change unless there is a good reason for this to happen. There are a number of legitimate reasons for fluents to change value, such as action occurrences where the effects of the action change fluent values, or causal dependencies between fluents where changes in some fluents force changes in others. In the TAL formalism, all such legitimate reasons for change are represented explicitly using the reassignment macros R, I and X in dependency constraints and action type definitions. When translated, these statements result in constraints on the *Occlude* predicate.

In the logical theory, we want to formally encode the assumption that these are the *only* reasons for fluents to be occluded. This can be achieved using a special form of circumscription (McCarthy, 1980) called filtered circumscription (Doherty & Łukaszewicz, 1994) which involves adding a second-order formula to the narrative logical theory.

The formal definition of TAL circumscription policy will use the following terminology:

- Let $\mathcal{N}$ denote the collection of narrative statements contained in a narrative in $\mathcal{L}(\mathrm{ND})$, and let $\mathcal{N}_{\mathsf{per}}, \mathcal{N}_{\mathsf{obs}}, \mathcal{N}_{\mathsf{occ}}, \mathcal{N}_{\mathsf{acs}}, \mathcal{N}_{\mathsf{domc}}$, and $\mathcal{N}_{\mathsf{depc}}$ denote the sets of persistence statements, observation statements, action occurrence statements, action type specifications, domain constraint statements, and dependency constraint statements in $\mathcal{N}$, respectively.

- Let Γ denote the translation of $\mathcal{N}$ into $\mathcal{L}(\mathrm{FL})$ using the *Trans* translation function, and let $\Gamma_{\mathsf{per}}, \Gamma_{\mathsf{obs}}, \Gamma_{\mathsf{occ}}, \Gamma_{\mathsf{acs}}, \Gamma_{\mathsf{domc}}$, and Γ_{depc} denote the persistence formulas, observation formulas, action occurrence formulas, action type specifications, domain constraint formulas, and dependency constraint formulasin Γ, respectively.

- Let Γ_{fnd} denote the set of foundational axioms in $\mathcal{L}(\mathrm{FL})$, containing unique names axioms, unique values axioms, etc.

The *Occlude* predicate is circumscribed relative to the action definitions in Γ_{acs} and the dependency constraints in Γ_{depc} with all other predicates fixed, and *Occurs* is circumscribed relative to the action occurrence formulas in Γ_{occ} with all other predicates fixed. Due to structural constraints on $\mathcal{L}(\mathrm{ND})$ statements, as specified in the definitions of application formulas and balanced change formulas, quantifier elimination techniques can then be used to translate the two second-order circumscriptive theories into logically equivalent first-order theories (Doherty et al., 1998; Doherty, 1996), denoted by $\mathit{Circ}(\Gamma_{\mathsf{acs}} \wedge \Gamma_{\mathsf{depc}}; \mathit{Occlude})$ and $\mathit{Circ}(\Gamma_{\mathsf{occ}}; \mathit{Occurs})$, respectively.

The two resulting theories are combined and filtered with the $\mathcal{L}(\mathrm{FL})$ translations of the persistence statements in Γ_{per} (forcing persistent and durational fluents to adhere to the persistence or default value assumptions), the domain constraints in Γ_{domc}, and the observations and timing constraints in Γ_{obs}, yielding the theory $\Gamma' = \Gamma_{\mathsf{per}} \wedge \Gamma_{\mathsf{obs}} \wedge \Gamma_{\mathsf{domc}} \wedge \mathit{Circ}(\Gamma_{\mathsf{occ}}; \mathit{Occurs}) \wedge \mathit{Circ}(\Gamma_{\mathsf{depc}} \wedge \Gamma_{\mathsf{acs}}; \mathit{Occlude})$. Adding the $\mathcal{L}(\mathrm{FL})$ foundational axioms in Γ_{fnd} then yields the theory $\Delta = \Gamma' \wedge \Gamma_{\mathsf{fnd}}$.

The theory Δ is still a first-order theory, but lacks one important component: There is no formal characterization of the linear discrete temporal structure used by TAL. There are two alternatives: One can use an interpreted theory for the temporal structure, or an axiomatization can be added in the shape of a second-order theory Γ_{time} corresponding to the Peano axioms without multiplication.

The expression $\mathit{Trans}^{+}(\mathcal{N})$ will denote the result of translating the narrative $\mathcal{N}$ into $\mathcal{L}(\mathrm{FL})$ and applying this filtered circumscription policy. The $\mathcal{L}(\mathrm{ND})$ formula γ is preferentially entailed by the $\mathcal{L}(\mathrm{ND})$ narrative $\mathcal{N}$ iff $\mathit{Trans}^{+}(\mathcal{N}) \models \mathit{Trans}(\gamma)$.

Example 2.4.2 (Extended Hiding Turkey Scenario, continued)
The circumscription of the *Occurs* predicate in the action occurrences (**occ**) above (that is, $\mathit{Circ}(\Gamma_{occ}; \mathit{Occurs})$) is equivalent to the following first-order formula:

$$\forall s,t,a[\mathit{Occurs}(s,t,a) \leftrightarrow (s=1 \wedge t=4 \wedge a=\mathsf{Load}(\mathbf{G1})) \vee (s=5 \wedge t=6 \wedge a=\mathsf{Fire}(\mathbf{G1},\mathbf{T1}))]$$

The circumscription of the *Occlude* predicate in the action schemas (**acs**) and dependency constraints (**dep**) above (that is, $\mathit{Circ}(\Gamma_{\mathsf{depc}} \wedge \Gamma_{\mathsf{acs}}; \mathit{Occlude})$) is equivalent to the following set of first-order formulas:

$$\begin{aligned}&\forall t,\mathit{turkey}[\mathit{Occlude}(t,\mathsf{alive}(\mathit{turkey})) \leftrightarrow\\&\quad t=6 \wedge \mathit{turkey}=\mathbf{T1} \wedge \mathit{Holds}(5,\mathsf{loaded}(\mathbf{G1}),\mathbf{true}) \wedge \mathit{Holds}(5,\mathsf{hiding}(\mathit{turkey}),\mathbf{false})]\end{aligned}$$

$$\begin{aligned}&\forall t,\mathit{gun}[\mathit{Occlude}(t,\mathsf{loaded}(\mathit{gun})) \leftrightarrow\\&\quad \mathit{gun}=\mathbf{G1} \wedge (2 \le t \le 4 \vee t=6 \wedge \mathit{Holds}(5,\mathsf{loaded}(\mathbf{G1}),\mathbf{true}))]\end{aligned}$$

$$\forall t,\mathit{turkey}[\mathit{Occlude}(t,\mathsf{deaf}(\mathit{turkey})) \leftrightarrow \mathtt{false}]$$

$$\begin{aligned}&\forall t,\mathit{turkey}[\mathit{Occlude}(t,\mathsf{hiding}(\mathit{turkey})) \leftrightarrow\\&\quad \mathit{turkey}=\mathbf{T1} \wedge\\&\quad \exists t'[t=t'+1 \wedge \mathit{Holds}(t',\mathsf{hiding}(\mathit{turkey}),\mathbf{false}) \wedge \mathit{Holds}(t',\mathsf{deaf}(\mathit{turkey}),\mathbf{false}) \wedge\\&\qquad \mathit{Holds}(t',\mathsf{noise},\mathbf{true}) \wedge \exists u[t'=u+1 \wedge \mathit{Holds}(u,\mathsf{noise},\mathbf{false})]] \vee\\&\quad \mathit{turkey}=\mathbf{T1} \wedge\\&\quad \exists t'[t=t'+10 \wedge \forall\tau[t' \le \tau \le t'+9 \rightarrow \mathit{Holds}(\tau,\mathsf{hiding}(\mathit{turkey}),\mathbf{true}) \wedge \mathit{Holds}(\tau,\mathsf{noise},\mathbf{false})]]]\end{aligned}$$

$$\forall t[\mathit{Occlude}(t,\mathsf{noise}) \leftrightarrow 2 \le t \le 4]$$

■

Part II

Extensions to TAL

Chapter 3

Tackling the Qualification Problem using Fluent Dependency Constraints

Since the first version of PMON, the TAL logics have provided a flexible and powerful solution to the frame problem. Many aspects of the ramification problem were solved in PMON-RC. This chapter contains an article called *Tackling the Qualification Problem using Fluent Dependency Constraints*, written together with Patrick Doherty, which presents one TAL-based approach for dealing with the qualification problem, one of the three well-known problems within the area of reasoning about action and change. Except for removing those parts of the TAL description that were already presented in this thesis, reformatting some narrative statements to ensure a consistent style and fixing one or two minor typos, this article is unchanged from the final version published in Computational Intelligence (Kvarnström & Doherty, 2000a).

3.1 Abstract

In the area of formal reasoning about action and change, one of the fundamental representation problems is providing concise, modular and incremental specifications of action types and world models, where instantiations of action types are invoked by agents such as mobile robots, and provided the preconditions to the action are true, their invocation results in changes to the world model concomitant with the goal-directed behavior of the agent. One particularly difficult class of related problems, collectively called the qualification problem, deals with the need to find a concise, incremental and modular means of characterizing the plethora of

exceptional conditions which might qualify an action, but generally do not, without having to explicitly enumerate them in the preconditions to an action. We show how fluent dependency constraints together with the use of durational fluents can be used to deal with problems associated with action qualification using a temporal logic for action and change called TAL-Q. We demonstrate the approach using action scenarios which combine solutions to the frame, ramification and qualification problems in the context of actions with duration, concurrent actions, nondeterministic actions and the use of both boolean and non-boolean fluents. The circumscription policy used for the combined problems is reducible to the first-order case.

3.2 Introduction

The primary focus of research in the area of formal reasoning about action and change considers representation problems associated with an autonomous agent, such as a mobile robot (UGV) or an unmanned aerial vehicle (UAV), interacting with a highly complex and dynamic environment in which the agent behaves in a goal-directed manner. A primary goal of the research is to develop modeling and verification tools which can be used by engineers in the development of such agents and by the agents themselves, who require both representations of the environment and limitations of their behavior in the environment, in order to execute tasks to achieve goals. Due to the dynamic and causal nature of an agent's interaction with its environment, temporal logic formalisms are ideal candidates for world modeling, task and planning specification and causal reasoning. The use of temporal logic formalisms provides a suitable basis for both specifying and verifying the complex activity associated with agent interaction with complex environments.

When focusing on the type of complex environments associated with UGVs and UAVs, it immediately becomes clear that it is in general computationally, epistemologically, and ontologically infeasible to completely represent the environment an agent is embedded in and the action types it has at its disposal when interacting with its environment. This leads to the use of nonmonotonic extensions to temporal formalisms which contribute to providing succinct and modular representations of incomplete world model specifications and action type specifications. This article will focus on the representation of action type specifications and agent task representations in terms of narratives. In our approach, narratives consist of different classes of statements, which include action type specifications, timed action occurrences, observations, domain and dependency constraints, and additional timing information relating statements to each other. Narratives can be viewed as agent programs to be executed by an agent, or as hypothetical courses of action an agent can reason about when generating its own plans, or simply trying to understand how its future actions will affect its external environment, or to what degree its past actions have achieved its previous goals.

Three difficult modeling problems associated with the formal specification of action types in the context of complex, dynamic environments are the *frame*, *ramification*, and *qualification* problems. These problems have been a topic of continual research in the action and change community. Briefly, the frame problem concerns the need to find a concise and efficient means of representing and reasoning about what does not change when an action or actions are executed by an agent. The ramification problem concerns the need to separate the representation of the direct effects of an action type description from the plethora of indirect effects that may ensue when the action is executed successfully. An important aspect of the problem is to deal with the context dependent and causal nature of the chains of indirect effects which may ensue. The qualification problem, which is the problem we will focus on in this article, concerns the need to find a concise, incremental and modular means of characterizing the plethora of exceptional conditions which might qualify an action, but generally do not, without having to explicitly enumerate them in the precondition to the action type. A solution to one of these problems generally implies a solution to the other two due to the interactions between preconditions, postconditions, and indirect effects of action occurrences.

Ascertaining whether one has solutions to each of these problems is as difficult as finding the solutions themselves. The reason for this is due to the fact that solutions that may work well when based on a particular set of assumptions regarding the ontological nature of the environment an agent is embedded in and particular epistemological constraints placed on the agent itself, may not work well when these assumptions and constraints are relaxed. Rather than there being one frame, ramification and qualification problem, we would claim that there are different solutions for different combinations of epistemological and ontological assumptions. This working hypothesis is well in-line with the approach used by Sandewall (1994) in his study of the frame problem using the Features and Fluents framework.

For example, some ontological assumptions concerning action types are whether actions with duration, non-deterministic actions, or concurrent actions are possible. An epistemological assumption would be whether an agent has complete knowledge about all the effects of an action, or whether one can assume complete and accurate sensory data about the environment. An additional factor when evaluating a solution pertains to what types of reasoning tasks one has in mind for the agent. If one is concerned with a predictive mechanism for the agent used when generating a plan, a solution to the qualification problem which works here might not work if one is concerned with a postdictive mechanism for the agent used after executing a number of actions in a plan and gathering sensor data about the results.

In this article, we will first informally discuss some of the different ontological and epistemological choices that may affect the nature of solutions to the qualification problem. We will then present a complex narrative description, the *Russian Airplane Hijack Scenario* (RAH), which in order to be adequately represented in any logical formalism, would require robust solutions to the frame, ramification and qualification problems. We say robust because a description of the RAH world requires the representation of concurrent actions, incomplete specification of states, ramification with chaining, the use of non-boolean fluents, fine-grained dependencies among objects in different fluent value domains, actions with duration, two types of qualification (*weak* and *strong*) and the use of explicit time, in addition to other features. To our knowledge, this provides one of the more challenging benchmark examples in the literature. It is challenging in the sense that it involves solutions to all three representation problems and the ontological assumptions pertaining to allowable action types are relatively complex.

The RAH narrative description will be used as a vehicle for considering different facets of the qualification problem and demonstrating our solutions to the problem. To do this, we will first introduce TAL-Q (Temporal Action Logic with Qualification), an extension to the already existing TAL family of logics (Doherty et al., 1998) which has sufficient expressivity to model the RAH scenario. TAL-Q is an incremental extension of an earlier logic called TAL-C (Karlsson & Gustafsson, 1999), just as TAL-C is an incremental extension of TAL-RC (Gustafsson & Doherty, 1996). In fact, the logical language and minimization policy is roughly the same for TAL-RC, TAL-C, and TAL-Q. The advantages of leaving the logic and minimization policy intact are that the new class of narrative descriptions that can be represented in TAL-Q subsumes previous classes and that any circumscribed scenario in TAL-Q is provably and automatically reducible to a first-order theory implemented in an on-line research tool developed by our group called VITAL (Kvarnström, 2005), which permits the visualization and querying of narrative descriptions.

After introducing TAL-Q, we will use it to represent the RAH narrative description. This will be done in stages. Initially, we will represent the narrative under the assumption that actions always succeed. We will then modify the representation with qualification conditions for action types and a mechanism for reasoning about qualified action types based on the use of durational fluents and dependency constraints. The use of durational fluents in combination with a simple form of circumscription provides a flexible means for incorporating a default mechanism into TAL-Q.

We will then use TAL-Q to consider a number of additional aspects pertaining to qualification in the context of different ontological choices such as the use of concurrent actions. In addition, we will briefly consider two alternative approaches to qualification that can be represented using TAL-Q. Finally, we will direct our attention toward a number of benchmark examples in the literature, representing them using TAL-Q, and then compare our approach to qualification with a number of other approaches in the literature.

3.3 The Qualification Problem

Before it is possible to design or assess any approach to solving the qualification problem, we must define in more detail what the qualification problem is.

Let us assume that there is an environment, a "real world", in which actions can be executed by one or more agents. Let us also assume that each action has a well-defined *intended effect*. For example, in the well-known blocks world, the intended effect of the action putdown(x) is that in the resulting state, the block x the agent is currently holding should be on the table and all other blocks should be unaffected by the action.

When reasoning about this world using a temporal action logic, we need a correct description of the preconditions and effects of each action type that can be used by an agent. If it is possible to find a model of the world that is both simple and correct, at least at some level of abstraction, it should be straightforward to find such a description of preconditions and effects of actions. However, in more complex worlds, describing an action may be far more difficult, and the resulting preconditions may be extremely complicated. This complexity is often due to a large number of conditions that are almost always false, but when satisfied, can cause the action to fail to achieve its intended effects. We will call such exceptional conditions *qualifications*, and if one or more of an action's qualifications hold, the action will be said to be *qualified*. (In some cases, even "non-exceptional" conditions will also be considered as qualifications.)

The potentially large number of qualifications to an action leads to a number of representational and implementational difficulties that are collectively called the *qualification problem*. Some of these difficulties will be discussed below.

3.3.1 Restricting the Problem

The qualification problem is a complex problem with many different aspects, and it would be very optimistic to assume that we can design a single solution that covers all these aspects. Instead, it is necessary to determine in advance which aspects of the problem should be addressed by the solution we are designing or assessing, which reasoning tasks (such as prediction or planning) should be supported by the solution, and which ontological and epistemological assumptions will be made regarding the worlds for which the solution should be applicable and the agents that will apply it. Below, we will consider these questions in some more detail.

Aspects of the Qualification Problem

Although the difficulties associated with the qualification problem are closely related, it is possible to isolate several aspects of the problem that may be tackled separately. Some of these aspects pertain to the following:

- Due to incomplete knowledge about the world one is reasoning about, it may be impossible, or at least very difficult, to find and enumerate all qualifications to an action. A classical example of this aspect of the qualification problem is the "potato in tailpipe" problem (Ginsberg & Smith, 1988): In order to start a car, there must be nothing wrong with the battery, there must be gas in the tank, there must not be a potato in the tailpipe, and so on. No matter how many conditions we manage to think of, there will surely always be more.

- Even when it is possible to know all qualifications to an action, the complexity of these conditions may require a highly expressive logic, unless we are willing to abstract away from some aspects of the world and be satisfied with incomplete specifications and a mechanism to deal with this incompleteness.
- The information we do have about the conditions under which an action is qualified needs to be represented in a modular manner, so that conditions may be added or removed incrementally.
- Assuming that actions are normally not qualified, the need to explicitly prove that each qualification condition does not hold may be computationally inefficient.

In this article, we will mainly concentrate on the representational problems associated with qualification, that is, modular and incremental representations of qualified action types.

Reasoning Tasks

In order to assess or design a solution to the qualification problem, we also need to specify the reasoning tasks that will be used by an agent in achieving goals via execution of actions. For example, an agent interested in determining why an action failed using postdiction may need a different solution than an agent that is solely interested in predicting the results of invoking a sequence of actions.

We will mainly consider off-line reasoning tasks such as prediction, postdiction and planning.

Ontological and Epistemological Assumptions

It is also necessary to determine which ontological assumptions will be made regarding the world in which the solution will be applied, as well as which epistemological assumptions will be made about the agent's knowledge of the world and of the effects of its actions. Perhaps the most important such assumption is that of what will happen in the world if the agent invokes a qualified action. The following are some of the assumptions that may be reasonable, depending on the world that is being modeled:

- Invoking a qualified action has no effect at all on the world.
- Invoking a qualified action affects the world, but we always know what effects it will have even when it is qualified.
- Invoking a qualified action affects the world in an unknown way, but only during the time interval when the action is being executed.

- Invoking a qualified action affects the world in an unknown way, and may trigger unknown chains of events that continue affecting the world after the action has finished executing.

However, there are also many other assumptions that may affect the applicability of a solution. The following are some examples of additional questions that need to be answered:

- Is there complete information about the initial state in a narrative? Is there any information about any other state in terms of observations made by the agent?
- Can actions be context-dependent? Can they be non-deterministic? Can they have duration, and if they can, do they have internal state (that is, may fluents change, discretely or continuously, within the duration of an action)? Can they have delayed effects? Can there be concurrent actions? If so, can actions overlap partially?
- Can there be dynamic processes continuously taking place independently of the actions invoked by the agent?
- In the presence of incomplete information and non-deterministic actions, are there domain constraints that exclude certain "impossible" states? Are there domain constraints that exclude certain "impossible" *sequences* of states? Can domain constraints vary over time?
- Are actions allowed to have indirect side effects? Can side effects be delayed (take place after the action has finished executing)? Can they trigger other side effects?

Clearly, the more complex the ontological and epistemological assumptions are, the more restricted our choices will be when attempting to solve the qualification problem for that particular class of worlds. Consequently, we need to determine these assumptions in advance.

Ideally, we would like to formally assess the correctness of different solutions to the qualification problem relative to a given class of narrative descriptions, specified via epistemological and ontological assumptions as Sandewall (1994) has done for the frame problem. However, extending Sandewall's framework for qualification – as well as ramification, concurrent actions, and other extensions we may want to use in combination with qualification – is outside the scope of this article. Instead, we will discuss in a more informal manner both some of the different questions that need to be considered when a solution to the qualification problem is designed and some of the effects the choice of reasoning task and our assumptions about the class of worlds we are reasoning about may have on the answers to these questions. We will then provide formal, but formally unassessed solutions using TAL-Q. Some of the existing solutions in the literature will also be considered from this point of view in Section 3.11.

3.3.2 Designing a Solution

We have now considered four questions: Which aspects of the qualification problem a solution should address, which reasoning tasks it should support, which ontological assumptions should be made regarding the worlds to which it is applicable, and which epistemological assumptions should be made regarding the agents that should apply it. For each of these questions, the answer will depend mainly on the class of problems we are trying to solve. For example, for anyone developing an agent controlling a UAV (unmanned aerial vehicle), the computational aspects of the qualification problem are very important, both prediction, postdiction and planning may be useful, and one must probably be able to model context-dependent concurrent actions with duration.

However, there are also certain design choices that may be made more or less independently of the problem or class of problems that should be solved. Some of these choices will be discussed in this section.

How should qualification conditions be expressed?

By definition, an action is qualified if it is somehow prevented from having its intended effects. There are basically two aspects to the problem. In an off-line mode, for example, when an agent is generating a plan, a predictive mechanism might simulate the possible future state of the world given that the agent executes a sequence or partially-ordered sequence of actions and find that the sequence violates certain domain or dependency constraints. In this case, either the domain or dependency constraints have been incorrectly specified, or the action type descriptions are not precise enough and a qualification condition for one or more actions has to be added. In an on-line mode, the agent actually executes sequences of actions and finds that one or several have not achieved their intended effects. This information is derived from actual sensory data. Since the world is its own model, either one has inaccurately specified the ontological assumptions which pertain to the world, or one of those rare qualifications has arisen and that qualification condition should be added to the agent's action type specification in an incremental manner so the next time the condition arises, the action will not be executed due to the explicit qualification. So, the qualification problem does not rule out adding a number of qualifications to an action, but any solution tries to minimize the number of explicit qualifications per action, and those that are added are added in a modular and incremental manner. Note that very little research has been done regarding the on-line execution and modification of action types. Most of the research has focused on generating the proper conclusions in off-line or simulation mode, assuming one already has explicit information about at least some of the qualifications per action, and on specifying a mechanism for adding new qualifications in an incremental and modular manner. In this article, we will also focus on the off-line mode.

Most formalisms for reasoning about action and change are based on the two-state assumption. There is an initial state in which an action is invoked and a result state in which the effects of the action become true provided the preconditions to the action are true in the initial state. TAL-Q is an exception due to its use of actions with duration and its use of explicit time. There are basically two classes of solutions in the literature, one focusing on the initial state of an action and the other focusing on the effect state.

When focusing on the initial state, one very straight-forward solution would be to strictly treat qualification constraints as preconditions to actions – conditions that must or must not hold in the state in which an action is invoked. For example, the start action can be considered qualified if potato-in-tailpipe is true in the state where an action will be executed. Most solutions in this class encode an assumption that if one can not explicitly prove that a known qualification to an action is true then that action can be executed. If actions may have duration and internal state, this approach can be extended by also allowing conditions at any time within the interval when the action is executed – for example, even if there was no potato in the tailpipe when the start action was invoked, one may be inserted during its execution.

In approaches focusing on the effect state, such as (Ginsberg & Smith, 1988; Lin & Reiter, 1994), an action is considered qualified whenever its intended effect would contradict a domain constraint in the effect state. In a sense, this implies a form of hypothetical reasoning which could only be used in off-line mode, where one checks whether the execution of an action would lead to a contradiction in the result state. In Ginsberg and Smith (1988), if this is the case, the action has no effect and the execution and effect states for the action are the same. In the case of Lin and Reiter (1994), a form of precompilation is used to modify the specified preconditions of each action to include the negation of every condition that would cause a contradiction. This assumes that one already has explicit qualification conditions in the theory. As before, this approach can of course be extended to actions with duration and internal state by considering an action qualified whenever it would contradict a domain constraint at any time during its execution. Also, if domain constraints can span multiple states (for example, relating fluents in one state to fluents in its successor), an action could be considered qualified whenever executing it must eventually result in such a domain constraint being contradicted.

We will pursue the precondition-based approach with TAL-Q, but with a much richer ontology of actions. This richer ontology would lead to problems in the latter approach. For example, if actions can be executed concurrently, it could be the case that the combined effect of two concurrent actions contradicts a domain constraint, but either action alone does not. Do we predict that one action succeeds, which may sometimes be the case, or that neither one does? Additional problems would arise if we allowed delayed side effects, non-deterministic actions, or any of a number of other features that have generally not been considered together with qualification. These problems make the latter approach much less intuitive for these extensions to the logics than it is for a situation calculus-type logic or belief-update approach described in Lin and Reiter (1994) and Ginsberg and Smith (1988). Due to the added expressivity of TAL-Q, these issues must be dealt with in our solution.

What should be entailed about the effects of invoking a qualified action?

In a number of formalisms, one can reason not only about the effects of actions that are unqualified, but also those that are qualified. In other words, an action is invoked even though not all conditions which would *guarantee* its intended effects are satisfied. Reasoning about this type of situation is perhaps more appropriate when considering the on-line mode, but still has to be defined even for off-line mode reasoning if the formalism allows invocation of qualified actions. Ideally, we would like our approach to the qualification problem to be able to represent whatever knowledge – or *lack* of knowledge – we have regarding the effects of invoking actions, whether qualified or not.

However, there are cases where this might not make sense, or is simply unnecessary. Suppose for instance, we have an interest in the planning task in off-line mode. In this case, reasoning about the effects of qualified actions does not make much sense, since the point to generating a plan is to generate a sequence of actions we assume are all executable and have their intended effects. What would be important is being able to reason about under what conditions an action might be qualified so as to avoid using it under those conditions in the plan generation phase. On the other hand, if one is using the formalism in on-line mode, reasoning about the effects of invoking qualified actions may be very important because an agent might on occasion invoke an action without being aware it is qualified – due to faulty sensors, for example. In this case, being able to reason about at least some of the effects of the action would be quite useful in a postdictive or diagnostic phase of reasoning.

Whatever choice is made, it should be made very clear why the choice is being made and what the ontological justification is. Quite often, the choice is simply a side effect of the solution chosen for solving the qualification problem. As we shall see in Section 3.11, many formalisms behave differently in this respect.

Should it be possible to reason about qualification within the logic?

One final design issue is whether qualifications to actions should be first-class objects which can be explicitly reasoned about in the formalism itself. This is particularly important in on-line reasoning mode, where execution monitoring is a central part of an agent's execution mechanism and determines future courses of action and modification of existing courses of action.

3.3.3 Reasoning about Undesirable Actions

A problem that often appears during planning is that of determining which actions would have effects that are undesirable. Although this may at a first glance seem unrelated to the qualification problem, it turns out that both problems can often be specified in terms of conditions that hold when an action is invoked or constraints that should not be violated by the effects of an action. Recall that this is the basis for the two classes of solutions to the qualification problem discussed previously. The difference between reasoning about qualified actions and reasoning about undesirable effects of actions may better be determined in terms of ontological assumptions placed on the worlds we are interested in. For example, should one make explicit distinctions between types of qualifications to actions such as those that if satisfied would make it physically impossible to execute the action satisfactorily, or those that simply involve contingent restrictions associated with the domain in question?

This appears to be the reason why some qualification examples in the literature are in fact examples of actions which *would* have their defined, intended, well-known effects, but which are invoked in a context in which those effects are not desirable. For example, in the lenient emperor scenario (Lin & Reiter, 1994), there is a robot that can paint blocks, but an emperor allows at most one block to be yellow at any given time. This is ensured by considering the action $\mathsf{paint}(block, \mathbf{yellow})$ to be qualified whenever there is already a yellow block (and, of course, by preventing the robot from executing any qualified action).

This approach works well when attempting to define a plan while avoiding undesirable actions. On the other hand, suppose that there is already a yellow block and that we want to predict what would happen if the robot tried to paint another block yellow. Certainly, there is nothing inherently problematic about this course of events even in the context of the emperor's strange rule. Intuitively, the action should succeed, with the conclusion that the robot invoked an action that violates correct social behavior. In the example above, the action is considered to be qualified and the conclusion will be that the action has in fact, failed.

Therefore, undesirable actions should probably not be handled in exactly the same way as qualified actions, but they can probably be handled in a technically very *similar* manner, and any solution to the qualification problem may also be interesting in this respect. Several examples in the literature which relate to this issue will be considered.

3.3.4 Summary

In summary, a solution to the qualification problem that works well for one reasoning task, under one ontological assumption, might not work well given another reasoning task or another ontological assumption, and when the set of problems one considers is extended, one may have to use a different approach previously considered less than optimal.

Due to these considerations, there is probably no single "best" solution to the qualification problem. Instead, there is likely to exist a *set* of good solutions, each of which is useful for a given expressivity and for a given task. Unfortunately, the solutions found in the literature often do not state explicitly what task and expressivity they are intended to handle. This makes it difficult to compare solutions, or build on one another's work. This section's intent was to point these issues out and create a context for the rest of the article. We will now consider the RAH scenario and its formalization in TAL-Q.

3.4 The Russian Airplane Hijack Scenario

In the remainder of this article, we will use the methodology of representative examples as a means of considering and proposing a solution to the qualification problem for a certain class of worlds. The scenario we will use is the Russian Airplane Hijack Scenario (RAH)[1], previously published in Doherty and Kvarnström (1998).

A Russian businessman, Boris, travels a lot and is concerned about both his hair and safety. Consequently, when traveling, he places both a comb and a gun in his pocket. A Bulgarian businessman, Dimiter, is less concerned about his hair, but when traveling by air, has a tendency to drink large amounts of vodka before boarding a flight to subdue his fear of flying. A Swedish businessman, Erik, travels a lot, likes combing his hair, but is generally law abiding.

Now, one ramification of moving between locations is that objects in your pocket will follow you from location to location. Similarly, a person on board a plane will follow the plane as it flies between cities.

Generally, when boarding a plane, the only preconditions are that you are at the gate and you have a ticket. However, if you try to board a plane carrying a gun in your pocket, which will be the case for Boris, this should qualify the action. Also, a condition that could sometimes qualify the boarding action is if you arrive at the gate in a sufficiently inebriated condition, as will be the case for Dimiter. When the boarding action is qualified, attempting to board should have no effect.

Boris, Erik and Dimiter already have their tickets. They start (concurrently) from their respective homes, stop by the office, go to the airport, and try to board flight SAS609 to Stockholm. Both Erik and Boris put combs in their pockets at home, and Boris picks up a gun at the office, while Dimiter is already drunk at home and may or may not already have a comb in his pocket. Who will successfully board the plane? What are their final locations? What will be in their pockets after attempting to board the plane and after the plane has arrived at its destination?

If the scenario is encoded properly and our intuitions about the frame, ramification and qualification problems are correct then we should be able to entail the following from the RAH scenario, assuming that Boris, Erik and Dimiter own the combs **comb1**, **comb2** and **comb3**, respectively:

1. **Erik** will board the plane successfully, eventually ending up at his destination.

2. An indirect effect of flying is that the person ends up at the same location as the airplane. In addition, because items in pockets follow the person, a transitive effect results where the items in the pocket are at the same location as the plane. Consequently, **Erik**'s comb **comb2** will also end up at his destination.

[1]This scenario is an elaboration and concretization of a sketch for a scenario proposed by Vladimir Lifschitz in on-line discussions in the Electronic Transactions on Artificial Intelligence (ETAI/ENAI).

3. **Boris** will get as far as the airport with a **gun** and **comb1** in his pocket. He will be unable to board the plane.

4. **Dimiter** will get as far as the airport, and may or may not be able to board the plane. If he is able to board the plane, he will eventually end up at his destination. Otherwise, he will remain at the airport. In any case, if he initially carried a comb, it will end up in the same location.

For this scenario, we assume that we know all possible reasons why an action may be qualified, and we are mainly interested in representing qualifications in a modular and intuitive manner. We are also mainly interested in prediction.

This is a rather complex scenario, and modeling it requires a relatively expressive logic. Unfortunately, many approaches to the qualification problem in the literature have been defined with very strong constraints placed on action types and also use the two-state assumption. Modeling this scenario in such logics, or scaling up the expressivity of such a logic to be able to model this scenario, would be difficult. In the next section, we will take advantage of the expressivity already part of TAL-Q in defining a solution.

3.5 TAL-Q: Temporal Action Logic with Qualification

Our approach to handling the qualification problem is based on the use of TAL-Q (Temporal Action Logic with Qualification), a member of the TAL (Temporal Action Logics) family of logics for reasoning about action and change.

As it turns out, the approach we will present does not require any new predicates or other changes to the high-level concepts used in previous TAL logics. Instead, it uses well-known concepts from older logics such as TAL-C (Karlsson & Gustafsson, 1999) in a new and different way. Therefore, we will begin by describing the logic TAL-Q without considering the qualification problem. In Section 3.6, we show how the RAH scenario can be modeled in TAL-Q under the assumption that actions never fail, and in Section 3.7, we define our approach to solving the qualification problem within TAL-Q and demonstrate it by applying it to the RAH scenario. *Since TAL-C has already been introduced in Chapter 2, the description of TAL-C in this section has been removed.*

3.6 Representing the RAH Scenario

In this section, we will show how the Russian Airplane Hijack Scenario can be represented in TAL-Q if we do not consider qualifications to actions. This will result in a scenario in which it is assumed that any attempt to board a plane always succeeds, regardless of whether the person carries a gun or is drunk. In Section 3.7, we will show how the scenario presented here can be modified in order to deal with the qualification problem.

In order to simplify the presentation, being at the airport will be the only normal precondition for boarding a plane.

All formulas in this section are written in the surface language $\mathcal{L}(\text{ND})$. Appendix 1 contains the same formulas, with the exception that the action type definitions have been modified as in Section 3.7 and some new dependency constraints have been added. Appendix 2 contains the translation of the formulas in Appendix 1 into the base logic $\mathcal{L}(\text{FL})$.

3.6.1 Narrative Background Specification

First, it is necessary to determine which value domains, fluents and actions are needed. For the Russian Airplane Hijack Scenario, we will need the standard boolean value domain boolean $= \{\mathbf{true}, \mathbf{false}\}$, a domain location $= \{\mathbf{home1}, \mathbf{home2}, \mathbf{home3}, \mathbf{office}, \mathbf{airport}, \mathbf{run609}, \mathbf{run609b}, \mathbf{air}\}$ for locations, and a domain thing $=$ $\{$ **gun**, **comb1**, **comb2**, **comb3**, **boris**, **dimiter**, **erik**, **sas609** $\}$ containing everything that has a location. We also define the subdomains runway $= \{\mathbf{run609}, \mathbf{run609b}\}$ for locations that are runways, plane $= \{\mathbf{sas609}\}$ for things that are airplanes, person $= \{\mathbf{boris}, \mathbf{dimiter}, \mathbf{erik}\}$ for things that are people, and pthing $= \{\mathbf{gun}, \mathbf{comb1}, \mathbf{comb2}, \mathbf{comb3}\}$ for things that people can pick up.

We also need four fluents: loc(thing) : location, inpocket(person, pthing) : boolean, onplane(plane, person) : boolean, and drunk(person) : boolean.

Four actions are necessary in this scenario: pickup(person, pthing) for picking up things, travel(person, location, location) for traveling between locations in the same city, board(person, plane) for boarding an airplane, and fly(plane, runway, runway) for flying between two runways.

Finally, we need to declare each of the four fluents persistent at all timepoints using a set of persistence statements:

per1 $\forall t, thing\ [\mathtt{true} \rightarrow Per(t+1, \mathsf{loc}(thing))]$
per2 $\forall t, person, pthing\ [\mathtt{true} \rightarrow Per(t+1, \mathsf{inpocket}(person, pthing))]$
per3 $\forall t, person\ [\mathtt{true} \rightarrow Per(t+1, \mathsf{drunk}(person))]$
per4 $\forall t, plane, person\ [\mathtt{true} \rightarrow Per(t+1, \mathsf{onplane}(plane, person))]$

3.6.2 Initial State

The initial state in a TAL narrative (as well as any other state) can be completely or incompletely specified using observation statements. For this scenario, we must define the initial locations of all things, as well as who is drunk in the initial state. On the other hand, we do not observe which things are in whose pockets.

obs1 $[0]\ \mathsf{loc}(\mathbf{boris}) \hat{=} \mathbf{home1} \wedge \mathsf{loc}(\mathbf{gun}) \hat{=} \mathbf{office} \wedge \mathsf{loc}(\mathbf{comb1}) \hat{=} \mathbf{home1} \wedge \neg\mathsf{drunk}(\mathbf{boris})$
obs2 $[0]\ \mathsf{loc}(\mathbf{erik}) \hat{=} \mathbf{home2} \wedge \mathsf{loc}(\mathbf{comb2}) \hat{=} \mathbf{home2} \wedge \neg\mathsf{drunk}(\mathbf{erik})$
obs3 $[0]\ \mathsf{loc}(\mathbf{dimiter}) \hat{=} \mathbf{home3} \wedge \mathsf{loc}(\mathbf{comb3}) \hat{=} \mathbf{home3} \wedge \mathsf{drunk}(\mathbf{dimiter})$
obs4 $[0]\ \mathsf{loc}(\mathbf{sas609}) \hat{=} \mathbf{run609}$

3.6.3 Action Definitions

Four actions were declared in the narrative background specification. The following action type specification defines the meaning of those actions. For example, if $\mathsf{fly}(plane, runway_1, runway_2)$ is invoked between t_1 and t_2, then assuming the airplane is initially at $runway_1$, it will be in the air in the interval (t_1, t_2) and finally end up at $runway_2$ at time t_2.

acs1 $[t_1, t_2]\ \mathsf{fly}(plane, runway_1, runway_2) \leadsto [t_1]\ \mathsf{loc}(plane) \mathrel{\hat{=}} runway_1 \rightarrow$
$I((t_1, t_2)\ \mathsf{loc}(plane) \mathrel{\hat{=}} \mathbf{air}) \wedge R([t_2]\ \mathsf{loc}(plane) \mathrel{\hat{=}} runway_2)$

acs2 $[t_1, t_2]\ \mathsf{pickup}(person, pthing) \leadsto [t_1]\ \mathsf{loc}(person) \mathrel{\hat{=}} value(t_1, \mathsf{loc}(pthing)) \rightarrow$
$R((t_1, t_2]\ \mathsf{inpocket}(person, pthing))$

acs3 $[t_1, t_2]\ \mathsf{travel}(person, loc_1, loc_2) \leadsto [t_1]\ \mathsf{loc}(person) \mathrel{\hat{=}} loc_1 \rightarrow R([t_2]\ \mathsf{loc}(person) \mathrel{\hat{=}} loc_2)$

acs4 $[t_1, t_2]\ \mathsf{board}(person, plane) \leadsto [t_1]\ \mathsf{loc}(person) \mathrel{\hat{=}} \mathbf{airport} \rightarrow$
$R([t_2]\ \mathsf{loc}(person) \mathrel{\hat{=}} value(t_2, \mathsf{loc}(plane)) \wedge \mathsf{onplane}(plane, person))$

The following action occurrences are also needed. The exact timepoints used below were not specified in the RAH scenario, but have been chosen arbitrarily. Alternatively, exact timepoints could have been avoided by using non-numerical temporal constants. Note, however, that many of the actions are concurrent, sometimes with partially overlapping intervals.

occ1	$[1,2]\ \mathsf{pickup}(\mathbf{boris}, \mathbf{comb1})$	**occ8**	$[7,9]\ \mathsf{travel}(\mathbf{erik}, \mathbf{office}, \mathbf{airport})$
occ2	$[1,2]\ \mathsf{pickup}(\mathbf{erik}, \mathbf{comb2})$	**occ9**	$[8,10]\ \mathsf{travel}(\mathbf{boris}, \mathbf{office}, \mathbf{airport})$
occ3	$[2,4]\ \mathsf{travel}(\mathbf{dimiter}, \mathbf{home3}, \mathbf{office})$	**occ10**	$[9,10]\ \mathsf{board}(\mathbf{dimiter}, \mathbf{sas609})$
occ4	$[3,5]\ \mathsf{travel}(\mathbf{boris}, \mathbf{home1}, \mathbf{office})$	**occ11**	$[10,11]\ \mathsf{board}(\mathbf{boris}, \mathbf{sas609})$
occ5	$[4,6]\ \mathsf{travel}(\mathbf{erik}, \mathbf{home2}, \mathbf{office})$	**occ12**	$[11,12]\ \mathsf{board}(\mathbf{erik}, \mathbf{sas609})$
occ6	$[6,7]\ \mathsf{pickup}(\mathbf{boris}, \mathbf{gun})$	**occ13**	$[13,16]\ \mathsf{fly}(\mathbf{sas609}, \mathbf{run609}, \mathbf{run609b})$
occ7	$[5,7]\ \mathsf{travel}(\mathbf{dimiter}, \mathbf{office}, \mathbf{airport})$		

3.6.4 Domain Constraints

We will define three domain constraints: No `pthing` can be carried by two `persons` at the same time, no `person` can be on board two `planes` at the same time, and any `pthing` in a `person`'s pocket must be at the same location as that `person`.

dom1 $\forall t, pthing, person_1, person_2$
$[person_1 \neq person_2 \wedge [t]\ \mathsf{inpocket}(person_1, pthing) \rightarrow [t]\ \neg\mathsf{inpocket}(person_2, pthing)]$

dom2 $\forall t, person, plane_1, plane_2$
$[plane_1 \neq plane_2 \wedge [t]\ \mathsf{onplane}(plane_1, person) \rightarrow [t]\ \neg\mathsf{onplane}(plane_2, person)]$

dom3 $\forall t, person, pthing\ [[t]\ \mathsf{inpocket}(person, pthing) \rightarrow [t]\ \mathsf{loc}(pthing) \mathrel{\hat{=}} value(t, \mathsf{loc}(person))]$

3.6.5 Dependency Constraints

Now, apart from qualifications, only the side effects of actions remain to be modeled: Anything on board an airplane should follow the airplane, and anything a person carries should follow the person. The following two dependency constraints are sufficient for achieving this. For example, if someone is on board a plane and the location of the plane changes to *loc*, the location of the person also changes to *loc*.

dep1 $\forall t, plane, person, loc$ [
$[t]$ $\mathsf{onplane}(plane, person) \wedge C_T([t]\ \mathsf{loc}(plane) \mathrel{\hat{=}} loc) \rightarrow R([t]\ \mathsf{loc}(person) \mathrel{\hat{=}} loc)]$

dep2 $\forall t, person, pthing, loc$ [
$[t]$ $\mathsf{inpocket}(person, pthing) \wedge C_T([t]\ \mathsf{loc}(person) \mathrel{\hat{=}} loc) \rightarrow R([t]\ \mathsf{loc}(pthing) \mathrel{\hat{=}} loc)]$

3.7 Representing the Qualification Problem in TAL-Q

We have now modeled most of the Russian Airplane Hijack Scenario in TAL-Q, but we have not yet taken care of the qualifications defined by the scenario: Someone who carries a gun cannot board a plane, and someone who is drunk may or may not be able to board.

There are already a number of solutions to various aspects of the qualification problem in the literature, some of which would be applicable to the TAL logics. However, many of these solutions are dependent on the two-state assumption with highly constrained action types. We would like to provide a solution that retains the following features of TAL:

- Any state, including the initial state, can be completely or incompletely specified using observations and domain constraints.
- Actions can be context-dependent and non-deterministic. They can have duration and internal state, and the duration may be different for different executions of the action. There may be concurrent actions with partially overlapping execution intervals.
- There can be dynamic processes continuously taking place independently of any actions that may occur.
- Domain constraints can be used for specifying logical dependencies between fluents generally true in every state or across states. They may vary over time.
- Actions can have side effects, which may be delayed and may affect the world at multiple points in time. They may in turn trigger other delayed or non-delayed side effects.

We would also like to retain the first-order reducibility of the circumscription axiom. The following restrictions and assumptions will apply. As discussed in Section 3.3.2, we will be satisfied with a solution where invoking a qualified action either has no effect or has some well-defined effect. We will also restrict the solution to the off-line planning and prediction problems, and not claim a complete solution for the postdiction problem, which would require being able to conclude that an action was qualified because its successful execution would have contradicted an observation of some fluent value after that action was invoked.

3.7.1 Enabling Fluents

To handle the qualification problem, we will propose a solution based on defaults where each action type in a narrative is associated with an *enabling fluent*, a boolean durational fluent with default value **true** and with the same number and type of arguments as the action type. This fluent will be used in the precondition of the action, and will usually be named by prefixing "poss-" to the name of the action. For example, the boarding action in the RAH scenario will be associated with an enabling fluent $\mathsf{poss\text{-}board}(person, plane)$. We add a persistence statement for this fluent and modify **acs4** as follows:

per5 $\forall t, person, plane\ [\mathtt{true} \to Dur(t, \mathsf{poss\text{-}board}(person, plane), \mathbf{true})]$

acs4′ $[t_1, t_2]\ \mathsf{board}(person, plane) \leadsto$
$[t_1]\ \mathsf{poss\text{-}board}(person, plane) \wedge \mathsf{loc}(person) \mathrel{\hat{=}} \mathbf{airport} \to$
$R([t_2]\ \mathsf{loc}(person) \mathrel{\hat{=}} value(t_2, \mathsf{loc}(plane)) \wedge \mathsf{onplane}(plane, person))$

The other action types are modified in a similar way (see Appendix 1 for more details).

Now, suppose that $\mathsf{board}(person, plane)$ is executed between timepoints t_1 and t_2. If $\mathsf{poss\text{-}board}(person, plane)$ is false at t_1 for some reason, the action is qualified, or *disabled*. On the other hand, if the fluent is true at t_1, the action is *enabled*. Of course, it can still be the case that the action has no effects, if other parts of its precondition are false.

To generalize this, a context-independent action that should have no effect at all when qualified can be defined using a simple action definition of the form[2]

acs*m* $[t_1, t_2]\ \mathsf{action} \leadsto [t_1]\ \mathsf{poss\text{-}action} \wedge \alpha \to R([t_2]\ \beta)$

where α is the precondition and β specifies the direct effects of the action (context-dependent actions are defined analogously). However, we also wanted to be able to define actions that do have some effects when they are qualified. This can be done by defining a context-dependent action that defines what happens when the enabling fluent is false:

acs*n* $[t_1, t_2]\ \mathsf{action} \leadsto ([t_1]\ \mathsf{poss\text{-}action} \wedge \alpha_1 \to R([t_2]\ \beta_1)) \wedge$
$([t_1]\ \neg\mathsf{poss\text{-}action} \wedge \alpha_2 \to R([t_2]\ \beta_2))$

For example, suppose that whenever anyone tries to board a plane but the action is qualified, they should be thrown in jail. In order to model this, we would add a new persistent fluent $\mathsf{in\text{-}jail}(\mathtt{person}) : \mathtt{boolean}$ and modify the boarding action from Section 3.6.3 as follows:

acs4″ $[t_1, t_2]\ \mathsf{board}(person, plane) \leadsto$
$([t_1]\ \mathsf{poss\text{-}board}(person, plane) \wedge \mathsf{loc}(person) \mathrel{\hat{=}} \mathbf{airport} \to$
$R([t_2]\ \mathsf{loc}(person) \mathrel{\hat{=}} value(t_2, \mathsf{loc}(plane)) \wedge \mathsf{onplane}(plane, person))) \wedge$
$([t_1]\ \neg\mathsf{poss\text{-}board}(person, plane) \wedge \mathsf{loc}(person) \mathrel{\hat{=}} \mathbf{airport} \to$
$R([t_2]\ \mathsf{in\text{-}jail}(person)))$

[2]Note that due to the regularity of the solution, such extensions could be implicit in an action macro, thus avoiding unneeded clutter in the representation.

In this alternative scenario, if anyone is at the airport and tries to board a plane, and the action is qualified, they will be thrown in jail. If they are at the airport but the action is not qualified, they will board the plane. If they are not at the airport, none of the preconditions will be true, and invoking the action will have no effect.

Regardless of whether a qualified action has an effect or not, its enabling fluent is a durational fluent with default value **true**. Therefore, the fluent will normally be true, and the action will normally be enabled. In the remainder of this section, we will examine some of the ways in which we can disable an action using strong and weak qualification.

3.7.2 Strong Qualification

Let us start with *strong* qualification. When an action is *strongly qualified*, it should definitely not succeed. This can be accomplished by forcing its enabling fluent to be false at the timepoint at which the action is invoked.

For example, suppose that when a person has a gun in his pocket, it should be impossible for that person to board a plane. Then, whenever $\mathsf{inpocket}(\mathsf{person}, \mathsf{gun})$ holds, we must make poss-board false. This can be achieved using a dependency constraint:

dep3 $\forall t, person, plane\ [[t]\ \mathsf{inpocket}(person, \mathbf{gun}) \rightarrow I([t]\ \neg\mathsf{poss\text{-}board}(person, plane))]$

At any timepoint t when a person has a gun in his pocket, we use the I macro both to occlude $\mathsf{poss\text{-}board}(person, plane)$ for all airplanes, thereby releasing it from the default value axiom, and to make it false. This implies that as long as a person has a gun in his pocket, poss-board will be false for that person on all airplanes. If the gun is later removed from the pocket, this dependency constraint will no longer be triggered. At that time, assuming no other qualifications affect the enabling fluent, it will automatically revert to its default value, **true**.

3.7.3 Weak Qualification

Although strong qualification can often be useful, we may sometimes want to express the fact that an action may succeed, or it may fail, depending on circumstances we may or may not be aware of. We call this *weak* qualification.

For example, we may want to model the fact that when a person is drunk, he *may or may not* be able to board an airplane, depending on whether airport security discovers this or not. We may not be able to determine within our model of the RAH scenario whether airport security does discover that any given person is drunk, and even if we could, it may be of no interest. In this case, whenever $\mathsf{drunk}(person)$ holds, we must release poss-board from the default value assumption:

dep4 $\forall t, person\ [[t]\ \mathsf{drunk}(person) \rightarrow X([t]\ \forall plane\ [\neg\mathsf{poss\text{-}board}(person, plane)])]$

At any timepoint t when a person is drunk, we occlude $\mathsf{poss\text{-}board}(\mathit{person}, \mathit{plane})$ for all airplanes, but since we do not state anything about the *value* of the enabling fluent, it is allowed to be either true or false.

Although being able to state that an action *may* fail is useful in its own right, it is naturally also possible to restrict the set of models further by adding more statements to the scenario, which could make it possible to infer whether $\mathsf{poss\text{-}board}$ $(\mathbf{dimiter}, \mathbf{sas609})$ is true or false at some or all timepoints. For example, we may know that people boarding **sas609** are always checked more carefully, so that it is impossible for anyone who is drunk to be on board that airplane, which could be expressed using a domain constraint. In the context of postdiction, observation statements could be used in a similar manner. For example, adding the observation statement **obs5** [13] $\mathsf{onplane}(\mathbf{sas609}, \mathbf{boris})$ to the narrative would allow us to infer that Boris did in fact board the plane and that $\mathsf{poss\text{-}board}(\mathbf{boris}, \mathbf{sas609})$ was in fact true. He would then end up at his intended destination. If instead we added the observation statement **obs6** [13] $\neg\mathsf{onplane}(\mathbf{sas609}, \mathbf{boris})$, we could infer that he was unable to board the plane and he did not end up at his destination.

The TAL-Q representation of the Russian Airplane Hijack Scenario from Section 3.4 is now complete. The full $\mathcal{L}(\mathrm{ND})$ narrative is listed in Appendix 1, and the translation into $\mathcal{L}(\mathrm{FL})$ is shown in Appendix 2.

The translation into $\mathcal{L}(\mathrm{FL})$ was done using VITAL (Kvarnström, 2005), a research tool that can be used to study problems involving action and change within TAL and generate visualizations of action scenarios and preferred entailments. VITAL was also used for generating Figure 3.1 on page 66 which is a color-coded summary of facts true in all preferred models of the RAH scenario. Light gray and dark gray stand for true and false values for boolean fluents. Medium gray stands for an unknown value, while black stands for a value which is unknown but will be the same as that of the previous timepoint due to inertia. For non-boolean fluents, "$*n*$" means that there are n possible values; the values are not shown in the diagram due to lack of space.

In this scenario, Dimiter is drunk at all timepoints, and he attempts to board a plane at time 9. There will be two classes of preferred models: In one class, Dimiter will successfully board the plane, and in the other, he will not. As shown in Figure 3.1, we can not infer $\mathsf{poss\text{-}board}(\mathbf{dimiter}, \mathbf{sas609})$ or its negation at any timepoint. In other words, we will not assume that the action succeeds merely because it is *possible* that it will succeed.

It should be noted that this approach has similarities to a standard default solution to the qualification problem, but with some subtle differences. For example, it permits more control of the enabling precondition, even allowing it to change during the execution of an action. More importantly, it involves no changes to the minimization policy already used in TAL to deal with the frame and ramification problems, and the circumscription policy inherits first-order reducibility.

3.8 Additional Aspects of the Qualification Problem

3.8.1 Qualification and Concurrency

One of the requirements we stated previously was that our solution should be able to handle concurrent actions. Here, there are two different cases, depending on whether the effects of the actions are independent or can interact in various ways. As we have seen when modeling the RAH scenario, the former case does not present a problem: Any number of people could attempt to board the plane at the same time, and the correct, intuitive conclusions would be obtained.

However, the latter case is far more interesting, and presents a problem for approaches where actions are qualified when their successful execution would contradict a domain constraint, due to the difficulties associated with determining exactly which of all concurrent actions was the cause of the contradiction. It is more easily handled with an approach where qualifications are conditions evaluated in the state where the action is invoked, such as our TAL-Q approach.

Assume, for example, that it is impossible for two people to board the same airplane at the same time (a resource limitation problem). Similar situations have already been considered in the context of TAL-C in Karlsson and Gustafsson (1999), where bounds on concurrency and limited resources were handled using fluent dependency constraints. In this approach, actions are decoupled from their effects using *influences*, boolean durational fluents that indicate that the world is inclined to change in some specific way, and a similar approach can be used for qualification. Below, we will show how the specific problem mentioned above can be modeled in TAL-Q using the same approach.

First, we add a new durational influence fluent $\mathsf{want\text{-}to\text{-}board}(\mathtt{person},\mathtt{plane})$ with default value **false**. We change the definition of board so that instead of altering the $\mathsf{onplane}$ fluent directly, the action simply makes $\mathsf{want\text{-}to\text{-}board}(\mathit{person},\mathit{plane})$ true at a single timepoint. Then, we add a new dependency constraint **dep5** that is triggered whenever $\mathsf{want\text{-}to\text{-}board}(\mathit{person},\mathit{plane})$ is true. This dependency constraint contains what were previously the direct effects of the action.

acs4‴ $[t_1,t_2]\ \mathsf{board}(\mathit{person},\mathit{plane}) \rightsquigarrow I([t_2]\ \mathsf{want\text{-}to\text{-}board}(\mathit{person},\mathit{plane}))$

dep5 $\forall t,\mathit{person},\mathit{plane}\ [[t]\ \mathsf{want\text{-}to\text{-}board}(\mathit{person},\mathit{plane}) \wedge \mathsf{poss\text{-}board}(\mathit{person},\mathit{plane}) \rightarrow$
$I([t]\ \mathsf{onplane}(\mathit{plane},\mathit{person}))]$

The scenario above is essentially a reformulation of the original RAH scenario, and will entail exactly the same facts. However, it is modeled using the TAL-C influence framework, which provides some additional flexibility in reasoning about actions and their effects. Specifically, there is now a simple way to define what should happen when two people try to board the same plane at the same time. Clearly, for that airplane, $\mathsf{want\text{-}to\text{-}board}(\mathit{person},\mathit{plane})$ will be true for more than one person, and we must make $\mathsf{poss\text{-}board}(\mathit{person},\mathit{plane})$ false for all except one of them. We add a new fluent $\mathsf{can\text{-}board}(\mathtt{plane}) : \mathtt{person}$ whose value at any given time is the unique person that can board the plane at that time. We then add two dependency constraints: One stating that if there is at least one person trying to board a certain plane *plane*, then can-board(*plane*) will be one of those people, and one stating that can-board(*plane*) is the only person who can board *plane*.

dep6 $\forall t, plane\ [\exists person\ [[t]\ \mathsf{want\text{-}to\text{-}board}(person, plane)] \rightarrow$
$\exists person_2\ [[t]\ \mathsf{want\text{-}to\text{-}board}(person_2, plane) \wedge I([t]\ \mathsf{can\text{-}board}(plane) \mathrel{\hat{=}} person_2)]]$

dep7 $\forall t, plane, person$
$[\neg([t]\ \mathsf{can\text{-}board}(plane) \mathrel{\hat{=}} person) \rightarrow I([t]\ \neg\mathsf{poss\text{-}board}(person, plane))]$

It is easy to imagine several variations on this problem. For example, if two or more people try to board a plane simultaneously, it could be the case that none of them should succeed, or that there should be priorities (the "strongest" one should succeed). This can easily be modeled by adapting other techniques presented in Karlsson and Gustafsson (1999).

3.8.2 Qualification: Not Only For Actions

As we have shown, this approach to qualification is based on general concepts already present in earlier TAL logics, such as durational fluents and fluent dependency constraints, instead of introducing new predicates, entailment relations or circumscription policies specifically designed for dealing with the qualification problem. This is appealing not only because we avoid introducing new complexity into the logic, but also because reusing these more general concepts adds to the flexibility of the approach. In this section, we will show how we can use exactly the same approach to specify qualifications not only for actions but for any generic rule or constraint.

Qualifying Qualification Constraints

When we initially considered the boarding action, the "natural" preconditions were that one had to be at the airport; this is the precondition encoded in the definition of board (**acs4**). Later, we found another condition that should qualify the action: No one should be able to board a plane carrying a gun. Now, however, we may discover that this qualification does not always hold: Airport security *should* be able to board a plane carrying a gun.

Assuming that there is a fluent is-security(`person`) : `boolean`, this exception to the general qualification rule could of course be modeled by changing the dependency constraint **dep3** in the following way:

dep3′ $\forall t, person, plane$
$[[t]\ \mathsf{inpocket}(person, \mathbf{gun}) \wedge \neg\mathsf{is\text{-}security}(person) \rightarrow I([t]\ \neg\mathsf{poss\text{-}board}(person, plane))]$

However, we may later discover additional conditions under which it should be possible for a person to board a plane with a gun, and we do not want to modify **dep3** each time. Instead, the qualification itself should be qualified. This can easily be done using the same approach as for actions. A new enabling fluent guns-forbidden(`person, plane`) : `boolean` is added for the qualification constraint, and **dep3** is modified as follows:

dep3″ $\forall t, person, plane\ [[t]\ \mathsf{inpocket}(person, \mathbf{gun}) \land \mathsf{guns\text{-}forbidden}(person, plane) \rightarrow$
$I([t]\ \neg \mathsf{poss\text{-}board}(person, plane))]$

Now, we can qualify the qualification **dep3** simply by making guns-forbidden false for some person and airplane. In order to do this, we add a new dependency constraint:

dep8 $\forall t, person, plane\ [[t]\ \mathsf{is\text{-}security}(person) \rightarrow I([t]\ \neg \mathsf{guns\text{-}forbidden}(person, plane))]$

Weakening Qualifications

It may also be the case that we want to qualify a strong qualification in order to "replace" it with a *weak* qualification. For example, suppose that a gun is made of a special kind of plastic that may or may not be detected by airport security. Assuming that we have already added dependency constraints **dep3″** and **dep8** as defined above, and that there is a fluent gun-is-plastic : `boolean`, we can achieve this in two different ways. First, we can use strong qualification for the guns-forbidden fluent, so that having a gun is definitely not a qualification to board, and then add a new weak qualification for the boarding action:

dep9 $\forall t, person, plane$
$[[t]\ \mathsf{inpocket}(person, \mathbf{gun}) \land \mathsf{gun\text{-}is\text{-}plastic} \rightarrow I([t]\ \neg \mathsf{guns\text{-}forbidden}(person, plane))]$

dep10 $\forall t, person, plane$
$[[t]\ \mathsf{inpocket}(person, \mathbf{gun}) \land \mathsf{gun\text{-}is\text{-}plastic} \rightarrow X([t]\ \neg \mathsf{poss\text{-}board}(person, plane))]$

Second, we can use weak qualification for the guns-forbidden fluent, so that having a gun may or may not qualify the boarding action:

dep11 $\forall t, person, plane$
$[[t]\ \mathsf{inpocket}(person, \mathbf{gun}) \land \mathsf{gun\text{-}is\text{-}plastic} \rightarrow X([t]\ \neg \mathsf{guns\text{-}forbidden}(person, plane))]$

Qualifying Dependency Constraints

As we have just shown, the same technique we used for qualifying actions could also be used for qualifying qualifications. Obviously, we could also apply the same technique to other parts of a narrative, such as ordinary dependency constraints. This allows us to express qualified side effects in TAL-Q, which we will demonstrate in Section 3.10.2.

3.8.3 Defining Enabling Fluents

In some approaches, qualification conditions are directly tied to specific actions, which can have certain advantages. For example, in our approach, it would have been possible to avoid the need to declare each enabling fluent and to explicitly include them in the corresponding action preconditions. This could be done by introducing a fixed $qualified(t, a)$ predicate expressing the fact that a specific action a is qualified at a timepoint t, and then modifying the translation of action type specifications from $\mathcal{L}(\mathrm{ND})$ into $\mathcal{L}(\mathrm{FL})$ in the appropriate manner. However, the fact that enabling fluents are ordinary fluents turns out to give us some additional flexibility in the way they are defined and used.

First, there is no strict requirement that a fluent must be *enabling*; we can also reverse its meaning and define a *disabling fluent*, if that is better suited for a particular scenario.

Second, there is of course also no formal requirement that the name of an enabling fluent is named by prefixing "poss-" to the name of the action – this is only a useful convention, which may be relaxed, especially when an enabling fluent is used for qualifying something other than an action.

Third, although our examples always associate a single unique enabling fluent with each action, it is possible to let multiple actions share the same enabling fluent, and one can also use multiple enabling fluents for the same action in order to model the fact that an action can be qualified for any of a set of possible reasons. This may be very useful when modeling larger scenarios. For example, if there is a robot that can move in four directions (actions move-north, move-south, move-east and move-west), and anything that makes the robot unable to move affects either all or none of these actions, we may want to use a single enabling fluent poss-move.

3.8.4 Interacting Qualifications

Since we are using two kinds of qualification – weak and strong – we must consider what will happen when an action is weakly and strongly qualified at the same time. By definition, this means that both $X([t]\ \neg\text{poss-action})$ and $I([t]\ \neg\text{poss-action})$ hold at the same timepoint t. But the X operator only releases the enabling fluent from the default value assumption, while the I operator both releases it and constrains its value – in this case, it forces poss-action to be false. In other words, the strong qualification takes precedence, and the action is strongly qualified.

3.8.5 Ramifications as Qualifications

Another problem related to the qualification problem occurs in formalisms where ramification constraints and qualification constraints are expressed as domain constraints (Ginsberg & Smith, 1988; Lin & Reiter, 1994). Assume, for example, that we are reasoning about the blocks world, and that we have the following domain constraint (expressed using TAL syntax), stating that no two blocks can be on top of the same block:

dom $\forall t, x, y, z\ [[t]\ \text{on}(x, z) \wedge \text{on}(y, z) \rightarrow x = y]$

Now, suppose that the direct effect of the action $\text{put}(\mathbf{A}, \mathbf{C})$ is $\text{on}(\mathbf{A}, \mathbf{C})$, and the action is executed in a state where $\text{on}(\mathbf{B}, \mathbf{C})$ is true. Then, we cannot determine syntactically whether the domain constraint should be interpreted as a ramification constraint (since no two blocks can be on top of **C**, **B** must be removed) or as a qualification constraint (since no two blocks can be on top of **C**, the action should fail).

In TAL-Q, however, all indirect effects of an action must be expressed as *directed* dependency constraints. Therefore, this problem simply does not arise. For example, if we want a ramification constraint, we can use the following dependency constraint:

dep $\forall t,x,y,z\ [[t]\ \mathsf{on}(x,z) \land C_T([t+1]\ \mathsf{on}(y,z)) \land x \neq y \rightarrow R([t+1]\ \neg\mathsf{on}(x,z))]$

If x is on z, and we then place y on z, then an indirect effect is that x is removed from z.

On the other hand, if we want a qualification constraint, we can introduce an enabling fluent $\mathsf{poss\text{-}put}(\mathtt{block},\mathtt{block})$ and add the following qualification condition:

dep $\forall t,x,y,z\ [[t]\ \mathsf{on}(x,z) \land x \neq y \rightarrow I([t]\ \neg\mathsf{poss\text{-}put}(y,z))]$

Clearly, the problem of determining whether a constraint should be interpreted as a qualification or a ramification does not arise in this approach.

3.9 Alternative Approaches to the Qualification Problem

We have now presented one approach to solving the qualification problem within the TAL framework, but this approach is certainly not the only one. Below, we will examine in somewhat less detail some alternative approaches.

3.9.1 Using Domain Constraints

Although our main approach to the qualification problem is based on qualifying an action whenever a condition holds in the state in which it is invoked, it is also interesting to investigate approaches based on qualifying an action whenever its execution would contradict a domain constraint.

One variation of this approach would involve simply adding the proper domain constraints to the scenario, and concluding that an action is qualified whenever the resulting narrative is inconsistent. For example, the constraint that no guns are allowed on board airplanes can be stated as follows:

dom4 $\forall t, plane\ [[t]\ \neg(\mathsf{loc}(\mathbf{gun}) \mathrel{\hat{=}} value(t, \mathsf{loc}(plane)))]$

Now, assume that we add this constraint to the initial version of the RAH scenario (from Section 3.6), where qualification was not considered. Since Boris tries to board the plane carrying a gun, we can infer that the gun will be on board the plane, but from **dom4** we can infer that no gun will ever be on board a plane, so the scenario is inconsistent, which means that some action must be qualified.

Obviously, this approach does not provide the correct conclusions about the results of invoking a qualified action, and due to the inconsistency it may not even seem like a solution at all. However, as discussed in Section 3.3.2, there are some cases where such approaches may still be useful, such as when we are doing planning. But even if this is the case, a more serious problem still occurs when this approach is used together with non-deterministic actions or incomplete information about the initial state. For example, suppose that Dimiter may (or may not) have a gun in the initial state. If he tries to board a plane, **dom4** will allow us to infer that he did *not* have a gun, when the intuitive conclusion would have been that the action may or may not be qualified.

3.9.2 Fault Fluents

By modifying the previous approach slightly, we can define another approach that may also have its uses. Instead of stating that a certain domain constraint must hold, we state that whenever it does *not* hold, a *fault fluent* should become true. For example, **dom4** from the previous section can be modified as follows:

dom4′ $\forall t, plane\ [[t]\ \mathsf{loc}(\mathbf{gun}) \hat{=} value(t, \mathsf{loc}(plane)) \rightarrow I([t]\ \mathsf{fault\text{-}gun\text{-}on\text{-}airplane})]$

Now, whenever someone is on board a plane and is carrying a gun, fault-gun-on-airplane will be true. From this, the agent can infer that an action must have been qualified.

This approach has several advantages. First, if some action is qualified, it is easier to find out which one and why it was qualified, since only the fault fluents need to be considered. Second, invoking a qualified action does not make the entire narrative inconsistent. And third, incomplete information and non-deterministic actions are not a problem. As before, suppose that Dimiter may (or may not) have a gun in the initial state. If he tries to board a plane, there will be two classes of models: One in which he did not have a gun and boards the plane without triggering the fault-gun-on-airplane fault fluent, and one where he did have a gun, boards the plane, and does make fault-gun-on-airplane true.

Unfortunately, the fault fluent approach still does not provide the correct conclusions if there is some qualified action, since it assumes that all actions succeed. However, there is another use for this approach, for which it appears to be perfectly suited. As mentioned in Section 3.3.3, qualification has sometimes been used for predicting whether the result of invoking a certain action would be *undesirable*. This has usually resulted in predicting that invoking an undesirable action is impossible or has no effect, when in reality, invoking the action would be possible and the action would have its undesirable effects.

Probably, what is needed for such scenarios is not the use of qualification but the use of a similar *mechanism* for providing "undesirability" conditions in an intuitive and modular way. That can be provided by the fault fluent approach, since it always predicts that an action succeeds, but can "flag" undesirable results by making a fault fluent true. For this task, the fault fluent approach would provide the correct results.

An interesting feature of this approach is that it can easily be combined with our main approach: True qualifications may be expressed as conditions holding in the invocation state, while undesirable results are expressed in terms of conditions that should hold in the resulting state.

3.10 Additional Examples

In this section, we will show how some qualification examples from the literature can be represented in TAL-Q, and we will also show an extension to one of those examples. Since the narrative type specifications are obvious from the examples, they will be omitted.

3.10.1 Dead Birds Don't Walk

We will begin with a relatively straightforward qualification example. There is a turkey, Fred, who can take walks. One constraint on this world is that it is not possible to walk when you are dead. Therefore, if Fred dies, one should conclude that he is no longer walking. On the other hand, if the walk action for Fred is invoked, we would intuitively want the action to be qualified – Fred should not suddenly become alive in order to satisfy the domain constraint (McCain & Turner, 1995).

In the TAL-Q representation of this scenario, we use the two boolean fluents alive and walking mentioned above, but we also need one enabling fluent per action type (**acs1**, **acs2**). If Fred is not alive, he can not be walking (**dom1**). A domain constraint is not adequate for inferring directed side-effects for actions. In this case, we use a dependency constraint (**dep1**) stating that when Fred dies (not at every timepoint where he is dead – note the use of C_T, "changes to true"), he stops walking. We also need a qualification (**dep2**) stating that when Fred is dead (C_T is not used), he cannot start walking. Together with the observation statement **obs1** and the action occurrences **occ1** and **occ2**, this allows one to infer that Fred is initially walking, then dies (and stops walking), and then cannot resume walking.

acs1 $[t_1, t_2]$ Die $\rightsquigarrow$ $[t_1]$ poss-die $\rightarrow R([t_2]\ \neg\text{alive})$
acs2 $[t_1, t_2]$ Walk $\rightsquigarrow$ $[t_1]$ poss-walk $\rightarrow R([t_2]$ walking$)$
dom1 $\forall t\ [[t]\ \neg\text{alive} \rightarrow \neg\text{walking}]$
dep1 $\forall t\ [C_T([t]\ \neg\text{alive}) \rightarrow R([t]\ \neg\text{walking})]$
dep2 $\forall t\ [[t]\ \neg\text{alive} \rightarrow I([t]\ \neg\text{poss-walk})]$
obs1 $[0]$ alive $\wedge$ walking
occ1 $[0, 1]$ Die
occ2 $[1, 2]$ Walk

3.10.2 A Simple Electric Circuit

Thielscher (1997) discusses qualified ramifications and presents a scenario in which there is an electric circuit with two batteries **bat1** and **bat2**, two switches **sw1** and **sw2**, and one light bulb. There is only one action, toggle(`switch`), whose only direct effect is that the given switch is toggled.

If you close switch **sw1**, the first battery is connected to the light bulb. Normally, this has the side effect that the light is turned on. But there are three qualifications to this ramification: The light is not turned on if the bulb is broken, if **bat1** is malfunctioning, or if the wiring is loose. Similarly, if you close switch **sw2**, the second battery is connected to the light bulb. Unfortunately, the voltage is too high, so usually, this will have the side effect that the bulb breaks. But here, there are also some qualifications: The bulb does not break if **bat2** is malfunctioning, or if the wiring is loose. Finally, there is normally no light when the bulb is broken.

Although our approach does not handle qualification for postdiction, we can easily handle the prediction problem for this scenario. One possible formalization is the following, using the persistent fluents $\mathsf{closed}(\mathtt{switch})$, light, broken, $\mathsf{malfunc}(\mathtt{battery})$ and $\mathsf{loose\text{-}wiring}$ and the enabling fluents $\mathsf{poss\text{-}light}$, $\mathsf{poss\text{-}break}$ and $\mathsf{no\text{-}light\text{-}when\text{-}broken}$.

obs1 $[0]\ \neg\mathsf{closed}(\mathbf{sw1}) \wedge \neg\mathsf{closed}(\mathbf{sw2})$
obs2 $[0]\ \neg\mathsf{broken} \wedge \neg\mathsf{loose\text{-}wiring} \wedge \forall \mathit{battery}\ [\neg\mathsf{malfunc}(\mathit{battery})]$
acs1 $[t_1, t_2]\ \mathsf{toggle}(\mathit{switch}) \rightsquigarrow ([t_1]\ \mathsf{closed}(\mathit{switch}) \rightarrow R([t_2]\ \neg\mathsf{closed}(\mathit{switch}))) \wedge$
$([t_1]\ \neg\mathsf{closed}(\mathit{switch}) \rightarrow R([t_2]\ \mathsf{closed}(\mathit{switch})))$
dom1 $\forall t\ [[t]\ \mathsf{no\text{-}light\text{-}when\text{-}broken} \rightarrow (\mathsf{broken} \rightarrow \neg\mathsf{light})]$
dep1 $\forall t\ [[t]\ \mathsf{poss\text{-}light} \wedge C_T([t]\ \mathsf{closed}(\mathbf{sw1})) \rightarrow R([t]\ \mathsf{light})]$
dep2 $\forall t\ [[t]\ \mathsf{poss\text{-}break} \wedge C_T([t]\ \mathsf{closed}(\mathbf{sw2})) \rightarrow R([t]\ \mathsf{broken})]$
dep3 $\forall t\ [[t]\ \mathsf{broken} \vee \mathsf{malfunc}(\mathbf{bat1}) \vee \mathsf{loose\text{-}wiring} \rightarrow I([t]\ \neg\mathsf{poss\text{-}light})]$
dep4 $\forall t\ [[t]\ \mathsf{malfunc}(\mathbf{bat2}) \vee \mathsf{loose\text{-}wiring} \rightarrow I([t]\ \neg\mathsf{poss\text{-}break})]$

3.10.3 Yellow Blocks are Forbidden

Returning once more to scenarios where only actions are qualified, we will now consider a scenario presented in Lin and Reiter (1994): A blocks world scenario where blocks may have different colors. There is a single robot which can paint blocks ($\mathsf{paint}(\mathtt{block}, \mathtt{color})$), but since yellow is traditionally reserved for the emperor, the robot is not allowed to paint any block yellow. In Lin and Reiter (1994), this is handled using qualification, by adding a domain constraint stating that no block may be yellow. Consequently, in that approach, the preconditions of the action $\mathsf{paint}(\mathit{block}, \mathbf{yellow})$ will always be false.

One possible translation to TAL-Q would use two domains, `block` and `color`, a fluent $\mathsf{col}(\mathtt{block}) : \mathtt{color}$ representing the color of a block, and an enabling fluent $\mathsf{poss\text{-}paint}(\mathtt{block}, \mathtt{color}) : \mathtt{boolean}$ with default value true, together with the following $\mathcal{L}(\mathrm{ND})$ statements. (Note that in Lin and Reiter (1994), all fluents will be undefined in the state resulting from invoking $\mathsf{paint}(x, \mathbf{yellow})$, while in our approach, the action will have no effect.)

obs1 $[0]\ \forall b\ [\neg(\mathsf{col}(b) \mathrel{\hat{=}} \mathbf{yellow})]$
acs1 $[t_1, t_2]\ \mathsf{paint}(b, c) \rightsquigarrow [t_1]\ \mathsf{poss\text{-}paint}(b, c) \rightarrow R([t_2]\ \mathsf{col}(b) \mathrel{\hat{=}} c)$
dep1 $\forall t, b\ [I([t]\ \neg\mathsf{poss\text{-}paint}(b, \mathbf{yellow}))]$

However, the fault fluent approach (Section 3.9.2) may be more appropriate, since it is more likely that the action would actually succeed, even though its effects were "illegal":

acs2 $[t_1, t_2]$ $\mathsf{paint}(b, c) \rightsquigarrow R([t_2]\ \mathsf{col}(b) \hat{=} c)$
dep2 $\forall t, b\ [[t]\ \mathsf{col}(b) \hat{=} \mathbf{yellow} \rightarrow I([t]\ \mathsf{fault\text{-}block\text{-}is\text{-}yellow}(b))]$

Using this approach, we will predict that painting a block yellow will succeed, but also that the fault fluent fault-block-is-yellow will become true for that block: We have performed an action that has undesirable results.

3.10.4 The Lenient Emperor

There is also a variation of the previous scenario in which the emperor is more lenient and allows at most one yellow block to exist. If we had thought ahead and provided an enabling fluent for **dep1** above, we could have handled this by qualifying the old qualification. Since we did not, we have to modify the existing qualification **dep1**. For example, it can be replaced with the following constraint:

dep1′ $\forall t\ [\exists b\ [[t]\ \mathsf{col}(b) \hat{=} \mathbf{yellow}] \rightarrow \forall b\ [I([t]\ \neg\mathsf{poss\text{-}paint}(b, \mathbf{yellow}))]]$

If we want to be able to paint a yellow block yellow again, we can use the following alternative:

dep1″ $\forall t\ [\exists b\ [[t]\ \mathsf{col}(b) \hat{=} \mathbf{yellow}] \rightarrow$
$\forall b\ [[t]\ \neg(\mathsf{col}(b) \hat{=} \mathbf{yellow}) \rightarrow I([t]\ \neg\mathsf{poss\text{-}paint}(b, \mathbf{yellow}))]]$

Again, the fault fluent approach may be more appropriate: If more than one block is yellow, we signal an error for each yellow block.

dep2′ $\forall t\ [\exists b_1, b_2\ [[t]\ \mathsf{col}(b_1) \hat{=} \mathbf{yellow} \land \mathsf{col}(b_2) \hat{=} \mathbf{yellow} \land b_1 \neq b_2] \rightarrow$
$\forall b\ [[t]\ \mathsf{col}(b) \hat{=} \mathbf{yellow} \rightarrow I([t]\ \mathsf{fault\text{-}block\text{-}is\text{-}yellow}(b))]]$

3.10.5 The Lenient Emperor – with Concurrency

An interesting variation of the lenient emperor scenario, which has not previously been considered in the literature, arises when there may be more than one agent in the world. For example, it may be the case that when no block is yellow, but a number of agents concurrently attempt to paint two or more blocks yellow, exactly one of them will succeed.

A similar scenario was discussed in Section 3.8.1, where at most one person could board a plane at any given timepoint. That, however, would be analogous to allowing at most one *new* yellow block at each timepoint. However, it turns out that the concurrent lenient emperor scenario can be modeled in a similar manner. First, we will reformulate the scenario using the TAL-C approach, using a durational influence fluent want-to-paint(`block`, `color`):

obs1 $[0]\ \forall b\ [\neg(\mathsf{col}(b) \hat{=} \mathbf{yellow})]$
acs1 $[t_1, t_2]\ \mathsf{paint}(b, c) \rightsquigarrow I([t_2]\ \mathsf{want\text{-}to\text{-}paint}(b, c))$
dep1 $\forall t, b, c\ [[t]\ \mathsf{want\text{-}to\text{-}paint}(b, c) \land \mathsf{poss\text{-}paint}(b, c) \rightarrow R([t]\ \mathsf{col}(b) \hat{=} c)]$

Although the influence fluent is not strictly necessary for this example, it can still be an advantage to model the scenario in this way due to the added flexibility in case the scenario ever needs to be changed. In this case, however, the important difference in the new scenario is that in **dep1**, poss-paint must be true at the *same* timepoint when the block should change color. This means that we only need to make sure that whenever any block is yellow, no block *except possibly that one* can be painted yellow. Note that this allows us to repaint a yellow block with the same color, and it also allows us to concurrently paint one block yellow and paint another, previously yellow block in another color.

dep2 $\forall t, b_1, b_2\ [[t]\ \mathsf{col}(b_1) \hat{=} \mathbf{yellow} \wedge b_1 \neq b_2 \rightarrow I([t]\ \neg\mathsf{poss\text{-}paint}(b_2, \mathbf{yellow}))]$

For this scenario, the fault fluent approach would be identical to that for the non-concurrent lenient emperor scenario.

3.11 Comparisons

Having considered some qualification examples and how they can be represented in TAL-Q, we will now compare our approach to some other approaches in the literature, beginning with McCarthy's introduction of circumscription (1980, 1986) and continuing with Lifschitz (1987), Shanahan (1997), Ginsberg and Smith (1988), Lin and Reiter (1994), McCain and Turner (1995), and finally Thielscher (1996a, 1996b).

Although these approaches have many differences, there are also many important similarities. Perhaps the most important of these similarities is that all of these approaches are based on the assumption that there is a single agent executing a simple sequence of actions without duration, and that all change in the world is caused by that agent. For example, there can be no concurrent actions, no delayed side effects, and no dynamic processes taking place in the background. Sometimes, not even non-deterministic actions are allowed. In other words, these approaches are not expressive enough to model the Russian Airplane Hijack Scenario.

For some of the approaches, it may be possible to extend them for more complex worlds without requiring major changes, in other words, a graceful scaling up. As we will see, however, several approaches are strongly dependent on the fact that actions and side effects can be represented as a function from the current state and the action to be performed to the successor state, or possibly the set of successor states. This is especially true for the approaches where qualification is based on constraints that must not be violated by an action, rather than on conditions that must or must not hold when the action is invoked (Ginsberg & Smith, 1988; Lin & Reiter, 1994).

3.11.1 McCarthy

McCarthy (1980) introduces circumscription and discusses how it can be used for conjecturing that any action will succeed unless there is something preventing its success. This is achieved using a $\mathit{prevents}(\mathit{reason}, \mathit{action}, \mathit{state})$ predicate which holds whenever some specific reason prevents an action from having its usual effects in the given state. Each such reason is then defined explicitly. For example, in a blocks world, we may say that heavy blocks cannot be moved: $\forall x, y, s.(\mathit{tooheavy}(x) \rightarrow \mathit{prevents}(\mathit{weight}(x), \mathit{move}(x, y), s))$. The *prevents* predicate is circumscribed relative to the conjunction of all such reasons, which allows us to predict that the action will succeed unless one of its qualifications holds when the action is invoked.

Clearly, this is very similar to the way we defined our main approach in Section 3.7. Like our approach, it can not be used for inferring qualifications based on observing that an action failed, since *prevents* is only circumscribed relative to the explicit qualification conditions. One important difference, however, is that our approach does not minimize qualifications – we minimize the occlusion predicate, which means that we minimize *potential* qualification. This is what allows us to express weak qualification.

In McCarthy (1986), a slightly different approach is used within the situation calculus. Instead of using a *prevents* predicate for qualification, a single *ab* ("abnormal") predicate is used for both qualification and many other tasks. The argument of *ab* is an *aspect*, an abstract object. For example, in the blocks world, we can move a block to a location unless the *move* action is abnormal in the first aspect: $\forall x, l, s.\ \neg \mathit{ab}(\mathit{aspect1}(x, \mathit{move}(x, l), s)) \rightarrow \mathit{loc}(x, \mathit{result}(\mathit{move}(x, l), s)) = l$. Then, we may not be able to lift heavy blocks: $\forall x, l, s.(\mathit{tooheavy}(x) \rightarrow \mathit{ab}(\mathit{aspect1}(x, \mathit{move}(x, l), s)))$. An interesting aspect of this approach is that if a qualified action is invoked, each fluent which would normally have been affected is released from the inertia assumption, but is not given a new value and is therefore allowed to vary freely. Fluents which would not have been affected retain their previous values.

3.11.2 Lifschitz: Formal Theories of Action

Unfortunately, as Lifschitz (1987) notes, global minimization of abnormality is not sufficient, since it sometimes leads to unintended models. He presents an alternative solution for the prediction task (or *temporal projection*), where it is assumed that all changes in the values of fluents are caused by actions.

Two new predicates are added to the situation calculus: $\mathit{causes}(a, p, f)$ expresses that the action a causes the primitive fluent p to have the same value that the fluent f had when the action was invoked, and $\mathit{precond}(f, a)$ expresses that the fluent f is a precondition to the action a. Given these new predicates, it is possible to define any number of preconditions to an action in an incremental manner. The situation-independent predicates *causes* and *precond* are then circumscribed, and an action is assumed to succeed iff all its preconditions hold when the action is invoked. If any of its preconditions do not hold, the action will be assumed to have no effect on the world.

This approach produced the correct results for the scenarios where McCarthy's earlier approach failed. However, apart from allowing more complex worlds to be modeled, the approach presented in Section 3.7 is also more flexible in the way qualification conditions can be specified. For example, our qualification conditions may vary over time, and may also depend on states other than the state in which the action is invoked. Due to the fact that enabling fluents are not directly tied to actions, we can also represent qualified qualifications and qualified side effects, while Lifschitz' approach does not allow side effects at all.

3.11.3 Shanahan: Solving the Frame Problem

Shanahan (1997) uses an approach similar to that of Lifschitz (1987), the main difference being that the *precond* predicate takes three arguments: $precond(f, v, a)$ expresses the fact that the action a is only executable when the fluent f has the value v. Consequently, the two approaches share many of the same advantages and disadvantages. As in Lifschitz' approach, if any precondition does not hold, an action will be assumed to have no effect on the world.

3.11.4 Ginsberg and Smith: Reasoning about Action II – The Qualification Problem

Ginsberg and Smith (1988) argue that specifying qualifications as preconditions to actions often leads to complicated formulas, due to the need to take all possible ramifications into account. Accordingly, they define a possible worlds approach in which each action is associated with a set of qualification constraints having the form of domain constraints. Given an action, the set of possible successors of the current world is first calculated without considering the qualification constraints. Then, any such world which does not satisfy all qualification constraints is discarded. If no possible successor remains, the action was qualified, and is assumed not to change the world at all.

This approach works very well for the examples examined by Ginsberg and Smith. However, if concurrent actions or delayed side effects were allowed, it would no longer be possible to reason about whether a single action would violate a domain constraint: For concurrent actions, it would be necessary to take into account all actions being performed at the same time, and it would be more difficult to determine exactly which action should be qualified. Similarly, if delayed side effects were allowed, one would have to know exactly which actions are invoked up to the time when the delayed side effect takes place. Qualified side effects would of course be even more problematic, since one would have to determine somehow whether it is the action itself or one of its side effects that should be qualified. In other words, this approach would be quite difficult to extend to handle complex scenarios such as the Russian Airplane Hijack Scenario.

3.11.5 Lin and Reiter: State Constraints Revisited

Lin and Reiter (1994) present a solution to the qualification problem within the situation calculus. The solution is based on generating an exact definition of the

$Poss(a,s)$ predicate, which states that it is possible to execute the action a in the state s. The definition of *Poss* is generated using both a set $\mathcal{D}_{nec}$ of formulas of the form $Poss(a,s) \supset \phi$ and a set $\mathcal{D}_{qual}$ of domain constraints that must hold in the state resulting from executing any action. The domain constraints in $\mathcal{D}_{qual}$ are regressed, and the results are combined with the formulas in $\mathcal{D}_{nec}$ to form an exact definition of *Poss*.

Since all qualification conditions are compiled into the definition of the *Poss* predicate, it is possible to infer that an action is qualified by evaluating *Poss* in the current situation. The situation $do(a,s)$ resulting from executing a qualified action a is completely undefined, since the successor state axioms only define fluent values in situations resulting from executing actions whose preconditions hold.

Like the approach used by Ginsberg and Smith, this solution also depends on the restricted expressivity of the logic being used – in fact, it does so to an even greater degree, due to the compilation of qualification conditions into a definition of *Poss*.

For example, if non-deterministic actions were allowed, we may only know that an action *may* contradict a domain constraint, so finding an exact definition of *Poss* would not be possible. Similarly, if actions with duration and internal state were introduced, the compilation procedure would be far more complicated due to the need to ensure that no intermediate state contradicts the domain constraints in $\mathcal{D}_{qual}$. If concurrent actions, delayed side effects or domain constraints referring to multiple states or domain constraints depending on time were allowed, this approach could not be used at all, since it would not be sufficient to consider the single action and situation used as arguments to the *Poss* predicate.

On the other hand, if the world one is reasoning about is simple enough, this solution does provide a way of specifying qualification constraints that is often more intuitive than using enabling fluents.

3.11.6 McCain and Turner: A Causal Theory of Ramifications and Qualifications

McCain and Turner (1995) provide a combined solution to the ramification and qualification problems in which every change must be *caused*. An action is qualified if it would imply a change that it did not cause. Causal laws are expressed using the form $\phi \Rightarrow \psi$ (if ϕ holds, ψ is caused to hold). Pure ramifications can be expressed using the form $True \Rightarrow \phi$, and pure qualifications using the form $\neg\phi \Rightarrow False$, but other forms of constraints can also be used.

A pure qualification $\neg\phi \Rightarrow False$ essentially defines a condition that must hold in any state resulting from executing an action. It does not cause fluents to change as a side effect of executing the action, but if the condition ϕ does not hold, *False* must hold in any resulting state – so there can be no resulting state, which means that the action was qualified. It is also possible to express "combined" ramification and qualification constraints. For example, the constraint $\neg Alive \Rightarrow \neg Walking$ may act as a ramification when *Alive* is caused to become false, but as a qualification when *Walking* is caused to become true.

Since the result of invoking a qualified action in this approach is an empty set of possible resulting states, it is not possible to reason about which value a fluent would take on after a qualified action was invoked – it is only possible to determine that the action would be qualified. Therefore, this approach is mainly useful for planning, or for prediction in the case where we are not interested in the result of invoking a qualified action.

If we consider an empty set of possible resulting states in this approach to be equivalent to an inconsistent scenario in the TAL formalism, any qualification constraint that can be expressed in this approach – pure or not pure – can also be expressed using our alternative approach from Section 3.9.1. Each qualification constraint becomes an ordinary domain constraint, while each ramification constraint is expressed as a fluent dependency constraint.

On an abstract level, this solution is very similar to the approaches used by Ginsberg and Smith (1988) and Lin and Reiter (1994), in the sense that pure qualifications are domain constraints that must not be violated by an action. The solution also has similar limitations in expressivity. However, the technical solution and the reasoning behind it are different, and so are the sets of possible states resulting from invoking a qualified action: In Ginsberg and Smith (1988) there was a single possible resulting state where nothing had changed, in Lin and Reiter (1994) the result was undefined, and in McCain and Turner (1995), there is no resulting state.

3.11.7 Thielscher: Causality and the Qualification Problem

Thielscher's approach to the qualification problem (1996a, 1996b) is quite different from the previous three approaches. In fact, it turns out to be more similar to the approach we have presented in Section 3.7, in that it uses fluents to represent qualifications. On the other hand, there are also quite a few differences.

Thielscher uses persistent *disqualification fluents*, which are assumed to be false in the *initial state* unless something forces them to be true, while our enabling fluents are durational, and are normally true in every state. While using durational fluents has the advantage of not needing to explicitly make a disqualification fluent false when a qualification no longer holds, Thielscher's approach has the advantage of being able to handle some cases of postdiction. For example, if the start action can only be qualified when potato is true, and if we observe that the action is qualified, then the conclusion would be that potato must have been true in the initial state.

This, of course, does not handle the case where we initially observed that there was no potato, then waited a while and tried to start the car, and starting failed. Thielscher handles this using *miraculous disqualification*, which allows an action to be qualified even though every explanation for its qualification is proven to be false, and which is globally minimized at a higher priority than ordinary qualification.

(Unfortunately, this also allows us to prove that there was in fact no potato.) There is also a method for qualifying ramification constraints within the same framework (Thielscher, 1997).

The result of executing a sequence $\langle a_1, \ldots, a_n \rangle$ of actions is a state defined by the function $Res(\langle a_1, \ldots, a_n \rangle)$, which is undefined when some action in the sequence is qualified. However, an observation of the form F `after` $\langle a_1, \ldots, a_n \rangle$ is still defined in this case, although it is always false for any formula F. This is yet another different definition of the state resulting from invoking a qualified action: Not even the tautology $\top$ is considered to hold.

But although this approach has certain advantages, it once again assumes a world where there are no concurrent actions, no actions with duration and internal state, no dynamic processes in the background, no delayed side effects, and no qualified side effects, and therefore, it would not be possible to model the Russian Airplane Hijack Scenario with this approach.

3.11.8 Summary

We have compared our approach to six other approaches in the literature. There turns out to be some similarities between all of these approaches, perhaps most importantly that they are designed for simple worlds in which a single agent performs actions in a sequential manner, and where any side effects – if allowed at all – take place in the ending state of the action. Therefore, none of these approaches are powerful enough to model the Russian Airplane Hijack Scenario, although for simpler scenarios, they sometimes provide more intuitive methods for specifying which actions are qualified.

There also appear to be two main approaches to the way in which qualification conditions are specified: Either as conditions holding in the initial state or as domain constraints that must not be violated by actions. Here, Thielscher's approach is an exception: It is possible to directly observe the qualification of an action, after which one can postdict the reasons for the qualification.

However, there is also one aspect in which the approaches are quite different: If we execute a qualified action, what can be said about the resulting state? Most approaches turn out to have their own answer to this question:

- In McCarthy (1986), the fluents that would ordinarily be affected by the action are released from the inertia assumption in the resulting state, but all other fluents remain inert. This could be emulated in TAL-Q using the X operator.
- In Lifschitz (1987), Shanahan (1997) and Ginsberg and Smith (1988), the action has no effect on the world. This is normally also the case in TAL-Q, unless an alternative effect has been specified.
- In Lin and Reiter (1994), the resulting state is completely undefined.

- In McCain and Turner (1995), there is no resulting state. This could be considered equivalent to an inconsistent scenario in TAL-Q.
- In Thielscher (1996a) and Thielscher (1996b), the result is a "state" in which *nothing* holds – not even a tautology.

3.12 Conclusion

We have presented an approach to the qualification problem based on the use of dependency constraints and durational fluents in the context of a highly expressive temporal logic of action and change called TAL-Q. TAL-Q permits the use of action types that are non-deterministic, context dependent, durational, and concurrent. This degree of expressivity introduces additional issues in solving the qualification problem not present in any of the previously proposed formalisms and solutions in the literature. We have also tried to show that whether any given approach to solving the qualification problem is useful or not often depends on both the reasoning task and the characteristics of the class of worlds we are interested in reasoning about. Although many solutions have been proposed in the literature, they often do not make such assumptions explicit, and often turn out to be useful only for a small class of worlds. The intent of this article was to present a solution to the qualification problem for TAL-Q in this context. Several of the ideas in the article are tentative and will be pursued in future research. One of the more important topics of research is to clarify the distinctions between on- and off-line reasoning modes and how these modes affect solutions to the qualification problem. In addition, pursuing the formal assessment of correctness for the proposed solutions to the qualification problem using TAL-Q is an important future research issue as are more formal comparative analyses of the alternative formalisms considered in the article.

Appendix 1: RAH Narrative in $\mathcal{L}(\mathrm{ND})$

For a narrative background specification for this scenario, see Section 3.6.1.

PERSISTENCE STATEMENTS

per1 $\forall t, thing\ [\mathtt{true} \rightarrow Per(t+1, \mathsf{loc}(thing))]$

per2 $\forall t, person, pthing\ [\mathtt{true} \rightarrow Per(t+1, \mathsf{inpocket}(person, pthing))]$

per3 $\forall t, person\ [\mathtt{true} \rightarrow Per(t+1, \mathsf{drunk}(person))]$

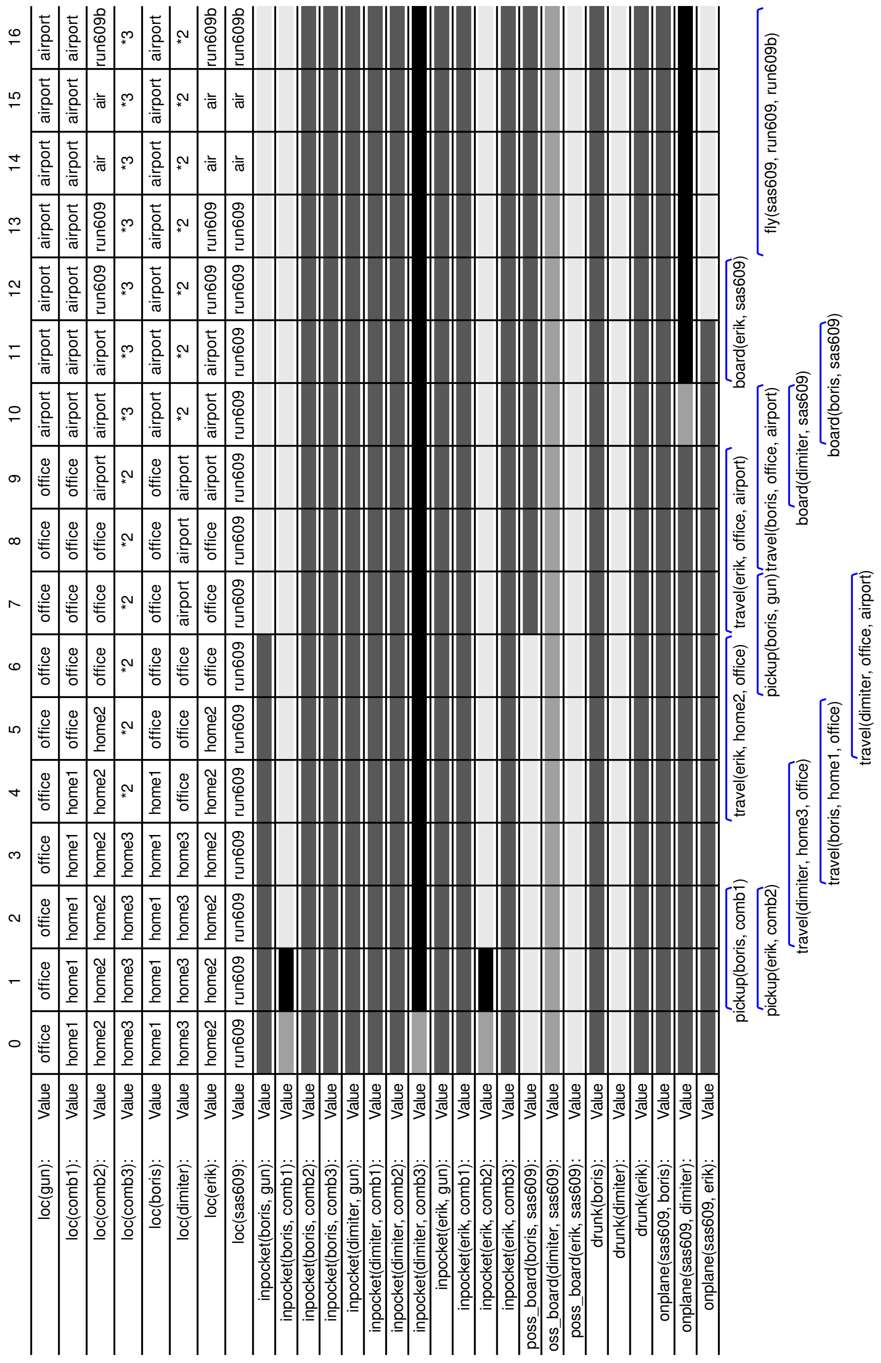
per4 $\forall t, plane, person\ [\mathtt{true} \rightarrow Per(t+1, \mathsf{onplane}(plane, person))]$

per5 $\forall t, person, plane\ [\mathtt{true} \rightarrow Dur(t, \mathsf{poss\text{-}board}(person, plane), \mathbf{true})]$

per6 $\forall t, person, pthing\ [\mathtt{true} \rightarrow Dur(t, \mathsf{poss\text{-}pickup}(person, pthing), \mathbf{true})]$

per7 $\forall t, person, loc_1, loc_2\ [\mathtt{true} \rightarrow Dur(t, \mathsf{poss\text{-}travel}(person, loc_1, loc_2), \mathbf{true})]$

per8 $\forall t, plane, runway_1, runway_2\ [\mathtt{true} \rightarrow Dur(t, \mathsf{poss\text{-}fly}(plane, runway_1, runway_2), \mathbf{true})]$



		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
loc(gun):	Value	office	office	office	office	office	office	office	office	office	office	airport	airport	airport	airport	airport	airport	airport
loc(comb1):	Value	home1	home1	home1	home1	home1	office	office	office	office	office	airport	airport	airport	airport	airport	airport	airport
loc(comb2):	Value	home2	home2	home2	home2	home2	home2	office	office	office	airport	airport	airport	run609	run609	air	air	run609b
loc(comb3):	Value	home3	home3	home3	home3	*2	*2	*2	*2	*2	*2	*3	*3	*3	*3	*3	*3	*3
loc(boris):	Value	home1	home1	home1	home1	home1	office	office	office	office	office	airport	airport	airport	airport	airport	airport	airport
loc(dimiter):	Value	home3	home3	home3	home3	office	office	office	airport	airport	airport	*2	*2	*2	*2	*2	*2	*2
loc(erik):	Value	home2	home2	home2	home2	home2	home2	office	office	office	airport	airport	airport	run609	run609	air	air	run609b
loc(sas609):	Value	run609	run609	run609	run609	run609	run609	run609	run609	run609	run609	run609	run609	run609	run609	air	air	run609b
inpocket(boris, gun):	Value	▓	▓	▓	▓	▓	▓	▓	░	░	░	░	░	░	░	░	░	░
inpocket(boris, comb1):	Value	▒	■	░	░	░	░	░	░	░	░	░	░	░	░	░	░	░
inpocket(boris, comb2):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓
inpocket(boris, comb3):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓
inpocket(dimiter, gun):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓
inpocket(dimiter, comb1):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓
inpocket(dimiter, comb2):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓
inpocket(dimiter, comb3):	Value	▒	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
inpocket(erik, gun):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓
inpocket(erik, comb1):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓
inpocket(erik, comb2):	Value	▒	■	░	░	░	░	░	░	░	░	░	░	░	░	░	░	░
inpocket(erik, comb3):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓
poss_board(boris, sas609):	Value	░	░	░	░	░	░	░	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓
oss_board(dimiter, sas609):	Value	▒	▒	▒	▒	▒	▒	▒	▒	▒	▒	▒	▒	▒	▒	▒	▒	▒
poss_board(erik, sas609):	Value	░	░	░	░	░	░	░	░	░	░	░	░	░	░	░	░	░
drunk(boris):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓
drunk(dimiter):	Value	░	░	░	░	░	░	░	░	░	░	░	░	░	░	░	░	░
drunk(erik):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓
onplane(sas609, boris):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓
onplane(sas609, dimiter):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▒	■	■	■	■	■	■
onplane(sas609, erik):	Value	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	░	░	░	░	░

pickup(boris, comb1)
pickup(erik, comb2)
travel(dimiter, home3, office)
travel(boris, home1, office)
travel(erik, home2, office)
travel(dimiter, office, airport)
pickup(boris, gun)
travel(erik, office, airport)
travel(boris, office, airport)
board(dimiter, sas609)
board(boris, sas609)
board(erik, sas609)
fly(sas609, run609, run609b)

Figure 3.1: Timelines for the Russian Airplane Hijack Scenario

OBSERVATIONS, ACTION OCCURRENCES AND TIMING

obs1 $[0]\ \mathsf{loc}(\mathbf{boris}) \hat{=} \mathbf{home1} \wedge \mathsf{loc}(\mathbf{gun}) \hat{=} \mathbf{office} \wedge \mathsf{loc}(\mathbf{comb1}) \hat{=} \mathbf{home1} \wedge \neg\mathsf{drunk}(\mathbf{boris})$

obs2 $[0]\ \mathsf{loc}(\mathbf{erik}) \hat{=} \mathbf{home2} \wedge \mathsf{loc}(\mathbf{comb2}) \hat{=} \mathbf{home2} \wedge \neg\mathsf{drunk}(\mathbf{erik})$

obs3 $[0]\ \mathsf{loc}(\mathbf{dimiter}) \hat{=} \mathbf{home3} \wedge \mathsf{loc}(\mathbf{comb3}) \hat{=} \mathbf{home3} \wedge \mathsf{drunk}(\mathbf{dimiter})$

obs4 $[0]\ \mathsf{loc}(\mathbf{sas609}) \hat{=} \mathbf{run609}$

occ1 $[1,2]\ \mathsf{pickup}(\mathbf{boris}, \mathbf{comb1})$

occ2 $[1,2]\ \mathsf{pickup}(\mathbf{erik}, \mathbf{comb2})$

occ3 $[2,4]\ \mathsf{travel}(\mathbf{dimiter}, \mathbf{home3}, \mathbf{office})$

occ4 $[3,5]\ \mathsf{travel}(\mathbf{boris}, \mathbf{home1}, \mathbf{office})$

occ5 $[4,6]\ \mathsf{travel}(\mathbf{erik}, \mathbf{home2}, \mathbf{office})$

occ6 $[6,7]\ \mathsf{pickup}(\mathbf{boris}, \mathbf{gun})$

occ7 $[5,7]\ \mathsf{travel}(\mathbf{dimiter}, \mathbf{office}, \mathbf{airport})$

occ8 $[7,9]\ \mathsf{travel}(\mathbf{erik}, \mathbf{office}, \mathbf{airport})$

occ9 $[8,10]\ \mathsf{travel}(\mathbf{boris}, \mathbf{office}, \mathbf{airport})$

occ10 $[9,10]\ \mathsf{board}(\mathbf{dimiter}, \mathbf{sas609})$

occ11 $[10,11]\ \mathsf{board}(\mathbf{boris}, \mathbf{sas609})$

occ12 $[11,12]\ \mathsf{board}(\mathbf{erik}, \mathbf{sas609})$

occ13 $[13,16]\ \mathsf{fly}(\mathbf{sas609}, \mathbf{run609}, \mathbf{run609b})$

ACTION TYPES

acs1 $[t_1, t_2]\ \mathsf{fly}(plane, runway_1, runway_2) \rightsquigarrow$
$[t_1]\ \mathsf{poss\text{-}fly}(plane, runway_1, runway_2) \wedge \mathsf{loc}(plane) \hat{=} runway_1 \rightarrow$
$I((t_1, t_2)\ \mathsf{loc}(plane) \hat{=} \mathbf{air}) \wedge R([t_2]\ \mathsf{loc}(plane) \hat{=} runway_2)$

acs2 $[t_1, t_2]\ \mathsf{pickup}(person, pthing) \rightsquigarrow$
$[t_1]\ \mathsf{poss\text{-}pickup}(person, pthing) \wedge \mathsf{loc}(person) \hat{=} value(t_1, \mathsf{loc}(pthing)) \rightarrow$
$R((t_1, t_2]\ \mathsf{inpocket}(person, pthing))$

acs3 $[t_1, t_2]\ \mathsf{travel}(person, loc_1, loc_2) \rightsquigarrow$
$[t_1]\ \mathsf{poss\text{-}travel}(person, loc_1, loc_2) \wedge \mathsf{loc}(person) \hat{=} loc_1 \rightarrow R([t_2]\ \mathsf{loc}(person) \hat{=} loc_2)$

acs4 $[t_1, t_2]\ \mathsf{board}(person, plane) \rightsquigarrow [t_1]\ \mathsf{poss\text{-}board}(person, plane) \wedge \mathsf{loc}(person) \hat{=} \mathbf{airport} \rightarrow$
$R([t_2]\ \mathsf{loc}(person) \hat{=} value(t_2, \mathsf{loc}(plane)) \wedge \mathsf{onplane}(plane, person))$

DOMAIN CONSTRAINTS

dom1 $\forall t, pthing, person_1, person_2$
$[person_1 \neq person_2 \wedge [t]\ \mathsf{inpocket}(person_1, pthing) \rightarrow [t]\ \neg\mathsf{inpocket}(person_2, pthing)]$

dom2 $\forall t, person, plane_1, plane_2$
$[plane_1 \neq plane_2 \wedge [t]\ \mathsf{onplane}(plane_1, person) \rightarrow [t]\ \neg\mathsf{onplane}(plane_2, person)]$

dom3 $\forall t, person, pthing\ [[t]\ \mathsf{inpocket}(person, pthing) \rightarrow [t]\ \mathsf{loc}(pthing) \hat{=} value(t, \mathsf{loc}(person))]$

DEPENDENCY CONSTRAINTS

dep1 $\forall t, plane, person, loc\ [[t]\ \mathsf{onplane}(plane, person) \wedge C_T([t]\ \mathsf{loc}(plane) \hat{=} loc) \rightarrow$
$R([t]\ \mathsf{loc}(person) \hat{=} loc)]$

dep2 $\forall t, person, pthing, loc\ [[t]\ \mathsf{inpocket}(person, pthing) \wedge C_T([t]\ \mathsf{loc}(person) \hat{=} loc) \rightarrow$
$R([t]\ \mathsf{loc}(pthing) \hat{=} loc)]$

dep3 $\forall t, person, plane\ [[t]\ \mathsf{inpocket}(person, \mathbf{gun}) \rightarrow I([t]\ \neg\mathsf{poss\text{-}board}(person, plane))]$

dep4 $\forall t, person, plane\ [[t]\ \mathsf{drunk}(person) \rightarrow X([t]\ \neg\mathsf{poss\text{-}board}(person, plane))]$

INTERMEDIATE SCHEDULE STATEMENTS

The following statements are generated from the action type specifications.[3]

scd1 $\forall t_1, t_2, plane, runway_1, runway_2.\ [t_1, t_2]\ \mathsf{fly}(plane, runway_1, runway_2) \rightarrow$
$([t_1]\ \mathsf{poss\text{-}fly}(plane, runway_1, runway_2) \wedge \mathsf{loc}(plane) \mathrel{\hat{=}} runway_1 \rightarrow$
$I((t_1, t_2)\ \mathsf{loc}(plane) \mathrel{\hat{=}} \mathbf{air}) \wedge R([t_2]\ \mathsf{loc}(plane) \mathrel{\hat{=}} runway_2))$

scd2 $\forall t_1, t_2, person, pthing.\ [t_1, t_2]\ \mathsf{pickup}(person, pthing) \rightarrow$
$([t_1]\ \mathsf{poss\text{-}pickup}(person, pthing) \wedge \mathsf{loc}(person) \mathrel{\hat{=}} value(t_1, \mathsf{loc}(pthing)) \rightarrow$
$R((t_1, t_2]\ \mathsf{inpocket}(person, pthing)))$

scd3 $\forall t_1, t_2, person, loc_1, loc_2.\ [t_1, t_2]\ \mathsf{travel}(person, loc_1, loc_2) \rightarrow$
$([t_1]\ \mathsf{poss\text{-}travel}(person, loc_1, loc_2) \wedge \mathsf{loc}(person) \mathrel{\hat{=}} loc_1 \rightarrow R([t_2]\ \mathsf{loc}(person) \mathrel{\hat{=}} loc_2))$

scd4 $\forall t_1, t_2, person, plane.\ [t_1, t_2]\ \mathsf{board}(person, plane) \rightarrow$
$([t_1]\ \mathsf{poss\text{-}board}(person, plane) \wedge \mathsf{loc}(person) \mathrel{\hat{=}} \mathbf{airport} \rightarrow$
$R([t_2]\ \mathsf{loc}(person) \mathrel{\hat{=}} value(t_2, \mathsf{loc}(plane)) \wedge \mathsf{onplane}(plane, person)))$

Appendix 2: RAH Narrative in $\mathcal{L}(\mathbf{FL})$

PERSISTENCE STATEMENTS

per1 $\forall t, thing, v\ [\neg Occlude(t+1, \mathsf{loc}(thing)) \rightarrow$
$(Holds(t, \mathsf{loc}(thing), v) \leftrightarrow Holds(t+1, \mathsf{loc}(thing), v))]$

per2 $\forall t, person, thing, v\ [\neg Occlude(t+1, \mathsf{inpocket}(person, thing)) \rightarrow$
$(Holds(t, \mathsf{inpocket}(person, thing), v) \leftrightarrow Holds(t+1, \mathsf{inpocket}(person, thing), v))]$

per3 $\forall t, person, v\ [\neg Occlude(t+1, \mathsf{drunk}(person)) \rightarrow$
$(Holds(t, \mathsf{drunk}(person), v) \leftrightarrow Holds(t+1, \mathsf{drunk}(person), v))]$

per4 $\forall t, plane, person, v\ [\neg Occlude(t+1, \mathsf{onplane}(plane, person)) \rightarrow$
$(Holds(t, \mathsf{onplane}(plane, person), v) \leftrightarrow Holds(t+1, \mathsf{onplane}(plane, person), v))]$

per5 $\forall t, person, plane\ [\neg Occlude(t, \mathsf{poss\text{-}board}(person, plane)) \rightarrow$
$Holds(t, \mathsf{poss\text{-}board}(person, plane), \mathbf{true})]$

per6 $\forall t, person, pthing\ [\neg Occlude(t, \mathsf{poss\text{-}pickup}(person, pthing)) \rightarrow$
$Holds(t, \mathsf{poss\text{-}pickup}(person, pthing), \mathbf{true})]$

per7 $\forall t, person, loc_1, loc_2\ [\neg Occlude(t, \mathsf{poss\text{-}travel}(person, loc_1, loc_2)) \rightarrow$
$Holds(t, \mathsf{poss\text{-}travel}(person, loc_1, loc_2), \mathbf{true})]$

per8 $\forall t, plane, runway_1, runway_2\ [\neg Occlude(t, \mathsf{poss\text{-}fly}(plane, runway_1, runway_2)) \rightarrow$
$Holds(t, \mathsf{poss\text{-}fly}(plane, runway_1, runway_2))]$

OBSERVATIONS, ACTION OCCURRENCES AND TIMING

obs1 $Holds(0, \mathsf{loc}(\mathbf{boris}), \mathbf{home1}) \wedge Holds(0, \mathsf{loc}(\mathbf{gun}), \mathbf{office}) \wedge$
$Holds(0, \mathsf{loc}(\mathbf{comb1}), \mathbf{home1}) \wedge \neg Holds(0, \mathsf{drunk}(\mathbf{boris}), \mathbf{true})$

obs2 $Holds(0, \mathsf{loc}(\mathbf{erik}), \mathbf{home2}) \wedge Holds(0, \mathsf{loc}(\mathbf{comb2}), \mathbf{home2}) \wedge$
$\neg Holds(0, \mathsf{drunk}(\mathbf{erik}), \mathbf{true})$

obs3 $Holds(0, \mathsf{loc}(\mathbf{dimiter}), \mathbf{home3}) \wedge Holds(0, \mathsf{loc}(\mathbf{comb3}), \mathbf{home3}) \wedge$
$Holds(0, \mathsf{drunk}(\mathbf{dimiter}), \mathbf{true})$

obs4 $Holds(0, \mathsf{loc}(\mathbf{sas609}), \mathbf{run609})$

occ1 $Occurs(1, 2, \mathsf{pickup}(\mathbf{boris}, \mathbf{comb1}))$

[3] In this variation of TAL-C, the *Occurs* predicate is not used. Instead, action type specifications are viewed as templates that are instantiated using action occurrence statements.

occ2 $Occurs(1, 2, \mathsf{pickup}(\mathbf{erik}, \mathbf{comb2}))$
occ3 $Occurs(2, 4, \mathsf{travel}(\mathbf{dimiter}, \mathbf{home3}, \mathbf{office}))$
occ4 $Occurs(3, 5, \mathsf{travel}(\mathbf{boris}, \mathbf{home1}, \mathbf{office}))$
occ5 $Occurs(4, 6, \mathsf{travel}(\mathbf{erik}, \mathbf{home2}, \mathbf{office}))$
occ6 $Occurs(6, 7, \mathsf{pickup}(\mathbf{boris}, \mathbf{gun}))$
occ7 $Occurs(5, 7, \mathsf{travel}(\mathbf{dimiter}, \mathbf{office}, \mathbf{airport}))$
occ8 $Occurs(7, 9, \mathsf{travel}(\mathbf{erik}, \mathbf{office}, \mathbf{airport}))$
occ9 $Occurs(8, 10, \mathsf{travel}(\mathbf{boris}, \mathbf{office}, \mathbf{airport}))$
occ10 $Occurs(9, 10, \mathsf{board}(\mathbf{dimiter}, \mathbf{sas609}))$
occ11 $Occurs(10, 11, \mathsf{board}(\mathbf{boris}, \mathbf{sas609}))$
occ12 $Occurs(11, 12, \mathsf{board}(\mathbf{erik}, \mathbf{sas609}))$
occ13 $Occurs(13, 16, \mathsf{fly}(\mathbf{sas609}, \mathbf{run609}, \mathbf{run609b}))$

SCHEDULE STATEMENTS

scd1 $\forall t_1, t_2, plane, runway_1, runway_2\ [Occurs(t_1, t_2, \mathsf{fly}(plane, runway_1, runway_2)) \rightarrow$
$(Holds(t_1, \mathsf{poss\text{-}fly}(plane, runway_1, runway_2)) \land Holds(t_1, \mathsf{loc}(plane), runway_1) \rightarrow$
$\forall t\ [t_1 < t \land t < t_2 \rightarrow Holds(t, \mathsf{loc}(plane), \mathbf{air}) \land Occlude(t, \mathsf{loc}(plane))] \land$
$Holds(t_2, \mathsf{loc}(plane), runway_2) \land Occlude(t_2, \mathsf{loc}(plane)))]$
scd2 $\forall t_1, t_2, person, pthing\ [Occurs(t_1, t_2, \mathsf{pickup}(person, pthing)) \rightarrow$
$(Holds(t_1, \mathsf{poss\text{-}pickup}(person, pthing)) \land$
$Holds(t_1, \mathsf{loc}(person), value(t_1, \mathsf{loc}(pthing))) \rightarrow$
$Holds(t_2, \mathsf{inpocket}(person, pthing), \mathbf{true}) \land Occlude(t_2, \mathsf{inpocket}(person, pthing)))]$
scd3 $\forall t_1, t_2, person, loc_1, loc_2\ [Occurs(t_1, t_2, \mathsf{travel}(person, loc_1, loc_2)) \rightarrow$
$(Holds(t_1, \mathsf{poss\text{-}travel}(person, loc_1, loc_2)) \land Holds(t_1, \mathsf{loc}(person), loc_1) \rightarrow$
$Holds(t_2, \mathsf{loc}(person), loc_2) \land Occlude(t_2, \mathsf{loc}(person)))]$
scd4 $\forall t_1, t_2, person, plane\ [Occurs(t_1, t_2, \mathsf{board}(person, plane)) \rightarrow$
$(Holds(t_1, \mathsf{poss\text{-}board}(person, plane), \mathbf{true}) \land Holds(t_1, \mathsf{loc}(person), airport) \rightarrow$
$Holds(t_2, \mathsf{loc}(person), value(t_2, loc(plane))) \land Holds(t_2, \mathsf{onplane}(plane, person), \mathbf{true}) \land$
$Occlude(t_2, \mathsf{loc}(person)) \land Occlude(t_2, \mathsf{onplane}(plane, person)))]$

DOMAIN CONSTRAINTS

dom1 $\forall t, pthing, person_1, person_2\ [person_1 \neq person_2 \land$
$Holds(t, \mathsf{inpocket}(person_1, pthing), \mathbf{true}) \rightarrow \neg Holds(t, \mathsf{inpocket}(person_2, pthing), \mathbf{true})]$
dom2 $\forall t, person, plane_1, plane_2\ [plane_1 \neq plane_2 \land Holds(t, \mathsf{onplane}(plane_1, person), \mathbf{true}) \rightarrow$
$\neg Holds(t, \mathsf{onplane}(plane_2, person))]$
dom3 $\forall t, person, pthing\ [Holds(t, \mathsf{inpocket}(person, pthing), \mathbf{true}) \rightarrow$
$Holds(t, \mathsf{loc}(pthing), value(t, \mathsf{loc}(person)))]$

DEPENDENCY CONSTRAINTS

dep1 $\forall t, plane, person, loc\ [Holds(t, \mathsf{onplane}(plane, person), \mathbf{true}) \land Holds(t, \mathsf{loc}(plane), loc) \land$
$\forall u\ [t = u + 1 \rightarrow \neg Holds(u, \mathsf{loc}(plane), loc)] \rightarrow$
$Holds(t, \mathsf{loc}(person), loc) \land Occlude(t, \mathsf{loc}(person))]$
dep2 $\forall t, person, pthing, loc\ [Holds(t, \mathsf{inpocket}(person, pthing), \mathbf{true}) \land$
$Holds(t, \mathsf{loc}(person), loc) \land \forall u\ [t = u + 1 \rightarrow \neg Holds(u, \mathsf{loc}(person), loc)] \rightarrow$
$Holds(t, \mathsf{loc}(pthing), loc) \land Occlude(t, \mathsf{loc}(pthing))]$
dep3 $\forall t, person, plane\ [Holds(t, \mathsf{inpocket}(person, \mathbf{gun}), \mathbf{true}) \rightarrow$
$\neg Holds(t, \mathsf{poss\text{-}board}(person, plane), \mathbf{true}) \land Occlude(t, \mathsf{poss\text{-}board}(person, plane))]$
dep4 $\forall t, person, plane\ [Holds(t, \mathsf{drunk}(person), \mathbf{true}) \rightarrow Occlude(t, \mathsf{poss\text{-}board}(person, plane))]$

TEMPORAL STRUCTURE AND FOUNDATIONAL AXIOMS

Apart from the narrative formulas above, we need axioms Γ_{time} for the temporal structure: The Peano axioms without multiplication. We also need the foundational axioms, Γ_{fnd}, which contain unique names axioms for the value sorts, fluent sorts, and actions. The foundational axioms also contain a set of axioms that relate the *Holds* predicate to the *value* function and ensure that each fluent has exactly one value at each timepoint:

$\forall t, thing, loc\ [Holds(t, \mathsf{loc}(thing), loc) \leftrightarrow value(t, \mathsf{loc}(thing)) = loc]$

$\forall t, person, pthing, v$
$[Holds(t, \mathsf{inpocket}(person, pthing), v) \leftrightarrow value(t, \mathsf{inpocket}(person, pthing)) = v]$

$\forall t, person, v\ [Holds(t, \mathsf{drunk}(person), v) \leftrightarrow value(t, \mathsf{drunk}(person)) = v]$

$\forall t, plane, person, v\ [Holds(t, \mathsf{onplane}(plane, person), v) \leftrightarrow value(t, \mathsf{onplane}(plane, person)) = v]$

$\forall t, person, plane, v$
$[Holds(t, \mathsf{poss\text{-}board}(person, plane), v) \leftrightarrow value(t, \mathsf{poss\text{-}board}(person, plane)) = v]$

$\forall t, person, pthing, v$
$[Holds(t, \mathsf{poss\text{-}pickup}(person, pthing), v) \leftrightarrow value(t, \mathsf{poss\text{-}pickup}(person, pthing)) = v]$

$\forall t, person, loc_1, loc_2, v$
$[Holds(t, \mathsf{poss\text{-}travel}(person, loc_1, loc_2), v) \leftrightarrow value(t, \mathsf{poss\text{-}travel}(person, loc_1, loc_2)) = v]$

$\forall t, plane, runway_1, runway_2, v\ [Holds(t, \mathsf{poss\text{-}fly}(plane, runway_1, runway_2), v) \leftrightarrow$
$value(t, \mathsf{poss\text{-}fly}(plane, runway_1, runway_2)) = v]$

Chapter 4

Elaboration Tolerance through Object-Orientation

TAL separates domain information into different statement classes such as observations, domain constraints and dependency constraints. This is sufficient for domains simple enough to be modeled in only a few logical statements, but for more complex domains, more structure is needed. This is especially true if one wants to build reusable domain models where individual aspects can be modified to suit new variations and new changes in the domain. This chapter contains an article called *Elaboration Tolerance through Object-Orientation*, which shows how object-oriented modeling can be applied in order to structure TAL narratives and facilitate reuse. This article was published in Artificial Intelligence (Gustafsson & Kvarnström, 2004), and like the article in the previous chapter, the only significant changes that have been made are related to reformatting the article and removing parts of the section describing the TAL logic.

4.1 Introduction

Traditionally, the semantic adequacy of formalisms for reasoning about action and change (RAC) has primarily been tested using very small specialized domains that highlight some particular point an author wants to make. These domains can usually be represented as a small number of simple formulas that are normally grouped by type rather than structure.

However, with some of the classical RAC problems completely or partially solved, and with powerful tools available for reasoning about action scenarios, it is now possible to model larger and more realistic domains. As soon as we start doing this, it becomes apparent that there is an unfortunate lack of methodology for handling complex domains in a systematic manner. There are few (if any) principles of good form, like the "No Structure in Function" principle from the qualitative reasoning community (de Kleer & Brown, 1984).

The following are some questions that must be answered in order to develop such a methodology:

Consistency: How can complex domains be modeled in a consistent and systematic way, to allow several developers to work on the same domain description and to enable others to understand the resulting domain more easily?

Elaboration tolerance (McCarthy, 1998): How do we ensure that domains can initially be modeled at a high level of abstraction, with the possibility to add further details at a later stage without completely redesigning the domain description? How do we design domain descriptions that can be modified in a convenient manner to take account of new phenomena or changed circumstances?

Modularity and reusability: How can particular aspects of a domain be designed as more or less self-contained modules? How do we provide support for reusing modules?

In this article, we investigate the applicability of the object-oriented paradigm (Abadi & Cardelli, 1996; Booch, 1991) to answering these questions. We model the entities that appear in a domain as *objects*, encapsulated abstractions that offer a well-defined *interface* to the surrounding world and hide the implementation-specific details. The interface consists of *methods* that can be called by other objects. Objects are instances of *classes* sharing the same *attributes* and methods. Classes are ordered in an *inheritance hierarchy* where a class can be created as a *subclass* of another class, inheriting the attributes and methods of the *superclass* and possibly adding its own attributes and methods or redefining some of the inherited methods.

Modeling entities as objects and interacting with them using methods provides a high degree of consistency in the domain model. The fact that attributes are hidden and accessed using methods increases elaboration tolerance, as does the ability to extend existing classes with new functionality in a structured and well-defined manner and to override existing functionality by re-implementing inherited methods. The modularity and reusability of a model are improved by modeling self-contained classes that are independent of the implementations of other classes.

The object-oriented concepts used in this article could potentially be applied to many different logics for reasoning and change, as long as they provide a certain minimum amount of expressivity. However, a proper demonstration of the viability of the approach requires a varied set of concrete examples. For these examples we have chosen to use a single logic: TAL-C (Karlsson & Gustafsson, 1999).

In the first part of the article, we will introduce TAL-C (Section 4.2), show how domains can be modeled in TAL-C in an object-oriented manner (Sections 5.3 and 5.4) and discuss some more complex issues related to object-orientation (Section 5.5) and how this affects elaboration tolerance (Section 5.6). Then, the ideas covered in the first part will be applied to the Missionaries and Cannibals domain

(Section 5.7). The 19 elaborations of this domain defined by McCarthy in his paper on elaboration tolerance (1998) will also be covered (Section 5.8), and a way of actually solving the problems within the logic is discussed (Section 5.9). An object-oriented model of the Traffic World domain (Sandewall, 1999) is briefly mentioned (Section 5.10). Finally, we conclude with related work (Section 5.11) and a discussion of the results (Section 5.12).

4.2 The TAL family and the TAL-C Logic

This article will use TAL-C as a basis for applying concepts from object-oriented modeling. A subset of this logic is implemented in the research tool VITAL (Kvarnström, 2005), a platform-independent Java tool that can be downloaded from the WWW. All narratives belonging to the subset supported by VITAL have a finite number of models, and VITAL uses constraint propagation techniques to generate all models (or any given number of models) of such narratives. This provides us with an experimental platform where object-oriented narratives can be tested.

Since TAL-C has already been introduced in this thesis, most of the description of TAL-C in this section has been removed. Please refer to Chapter 2 and the description of the Extended Hiding Turkey Scenario in Section 2.3.

One new macro was introduced in this section: The ***Set*** *macro, an alias for the interval reassignment macro previously called I.*

Apart from being more complex than many traditional benchmark problems in the RAC community (the even more well-known Stanford Murder Mystery requires only four short statements), the Extended Hiding Turkey Scenario presented in Section 2.3 is fairly representative of the area. The statements are ordered by type, with no special regard to the structure of the problem. The fluents are also unstructured in the sense that there is no indication that alive and hiding refer to properties of a turkey while loaded and noise do not.

Although the hiding turkey domain is still comprehensible in this unorganized form, it is clear that some additional structure will be valuable when modeling more complex domains. The following section presents a way of applying the object-oriented paradigm to modeling such domains.

4.3 Basic Object-Oriented Modeling in TAL-C

As has been shown previously (Karlsson & Gustafsson, 1999; Karlsson, Gustafsson, & Doherty, 1998; Kvarnström & Doherty, 2000a), the TAL logics are flexible and fine-grained logics suitable for handling a wide class of domains. We will now show how to use object-oriented modeling as a structuring mechanism for domain descriptions, thereby supporting the modeling of more complex domains and increasing the possibility of being able to reuse existing models when modeling related domains.

To simplify the task of the domain designer, some extensions to the $\mathcal{L}(\mathrm{ND})$ syntax will be introduced. These extensions are not essential, since the new macros and statement classes can mechanically be translated into the older syntax. The translations are implemented in the research tool VITAL.

The remainder of this section will show how classes are declared and how to instantiate objects of a specific class. We will then go on to discuss how to declare and use attributes (fields), and how to use three different types of methods: Accessors, mutators, and constraint methods. This provides the basic functionality for the object-oriented modeling of complex domains in TAL-C. Section 4.4 will cover additional topics such as how to override a method.

4.3.1 Defining Classes and Objects

In TAL, domains are traditionally modeled using an unstructured set of boolean or non-boolean fluents, each of which can take a number of arguments belonging to specific value domains.

In our object-oriented approach, we will instead concentrate on *classes* and *objects*. Each class will be modeled as a finite value domain, and each object as a value in that domain. Due to the order-sorted type structure used in TAL, inheritance hierarchies for classes are easily supported by modeling subclasses as subdomains. We will assume that the hierarchy has a single root called OBJECT.

Given the approach being used, it would be easy to introduce a class alias for the ordinary domain declaration statement. However, this would mean that any class declaration statement would have to explicitly enumerate all objects belonging to the class. Instead, a new, more flexible syntax is introduced which allows class and object declarations to be separated.

Defining Classes

The narrative type specification syntax in VITAL is extended to allow two forms of class declaration statement. A statement of the form class NEWCLASS declares a new *top-level class* named NEWCLASS, without a parent. Usually this is only used for the OBJECT class. A statement of the form class SUB *extends* SUPER declares a new *subclass* named SUB, with the parent class (superclass) SUPER. This makes SUB a *direct subclass* of SUPER, and SUPER is a *direct superclass* of SUB.

A class SUB is a *subclass* of SUPER iff it is a direct subclass of SUPER or there is an intermediate class INTER such that SUB is a direct subclass of INTER and INTER is a subclass of SUPER. The *superclass* concept is defined similarly.

A simple water tank domain will be used as a running example. This domain requires the standard root class OBJECT together with a domain TANK for water tanks. We are also interested in modeling a special type of tank, a FLOWTANK, which may have a flow of water into or out of the tank, as well as PIPEs between the tanks.

class OBJECT
class TANK *extends* OBJECT
class FLOWTANK *extends* TANK
class PIPE *extends* OBJECT

Defining Objects

Objects are declared in the narrative type specification using object statements (labeled obj). Declaring an object as a member of a class naturally also makes it a member of its superclasses: Any FLOWTANK is automatically also a TANK and an OBJECT.

obj **tank1** : TANK
obj **tank2**, **tank3** : FLOWTANK
obj **pipe1** : PIPE

Note that since classes correspond to value domains, it is possible to quantify over all objects belonging to a given class. Also note that objects are not created at any particular timepoint. They are declared in the narrative specification and exist at all timepoints.

Translation

Because VITAL requires a single definition for each value domain, class declarations and object declarations cannot be translated in isolation. Instead, the complete set of class and object declarations are translated into TAL-C in the following manner.

An object o is considered to be explicitly declared to belong to the class CL iff there is an object declaration statement having the following form:

obj $\ldots, o_i, \ldots$: CL

Each class declaration statement class NEWCLASS for a top-level class NEWCLASS is translated into the domain declaration statement domain NEWCLASS :elements $\{ o_1, \ldots, o_n \}$, where the objects $o_1, \ldots, o_n$ are exactly those objects that are explicitly declared to belong to NEWCLASS or to a subclass of NEWCLASS.

Each class declaration statement class NEWCLASS *extends* SUPER for a non-top-level class NEWCLASS is translated into the domain declaration statement domain NEWCLASS :extends SUPER :elements $\{ o_1, \ldots, o_n \}$, where the objects $o_1, \ldots, o_n$ are exactly those objects that are explicitly declared to belong to NEWCLASS or to a subclass of NEWCLASS.

This leads to the following VITAL domain definitions for the classes and objects declared above:

domain OBJECT :elements { **pipe1**, **tank1**, **tank2**, **tank3** }
domain TANK :elements { **tank1**, **tank2**, **tank3** }
domain FLOWTANK :elements { **tank2**, **tank3** }
domain PIPE :elements { **pipe1** }

4.3.2 Using Attributes

As usual in object-oriented languages, each object can be associated with a set of attributes, also known as fields. All objects of a certain class share the same attributes, but the specific values of the attributes may differ between the objects. Below, we show how attributes are modeled in TAL-C, how they are initialized, and how they can be changed at specific points in time.

Defining Attributes

All attributes are specified in attribute declarations (labeled `attr`). For example, any TANK has a current volume, a maximum volume, and a base area, all of which are `Real` values.[1] These attributes are persistent: They will not change unless explicitly changed. This is specified as follows:

attr TANK.volume : `Real`
attr TANK.maxvol : `Real`
attr TANK.area : `Real`

It is also possible to define attributes with arguments, which provides functionality similar to the use of arrays or mappings in programming languages. For example, if any water tank must keep track of exactly which pipes it is connected to, this can be modeled using a boolean attribute connected taking a pipe as an argument:

attr TANK.connected(PIPE) : `boolean`

An attribute is automatically translated into a fluent taking one additional argument – an object of the class to which the attribute belongs. Thus, the declarations above are translated into the TAL fluents volume(TANK) : `Real`, maxvol(TANK) : `Real`, area(TANK) : `Real`, and connected(TANK, PIPE) : `boolean`. Since time-dependent fluents are used, any attribute can vary over time in a natural manner.

More formally, an attribute declaration attr CLS.attr$(s_1, \ldots, s_n)$: s where $n \geq 0$ is translated into a feature declaration feature attr(CLS, $s_1, \ldots, s_n$) : s.

Using the standard TAL-C syntax, the volume attribute of **tank1** would be denoted by volume(**tank1**). To permit the use of the standard object-oriented syntax **tank1**.volume, we define $obj.\mathrm{attr}(x_1, \ldots, x_n) \overset{\mathbf{def}}{=} \mathrm{attr}(obj, x_1, \ldots, x_n)$, where $n \geq 0$; if $n = 0$, the parentheses may be omitted. This syntax will also be applied to method invocations.

Attributes in Subclasses

Due to the use of the order-sorted type structure in TAL-C, subclasses automatically inherit the attributes of their parents, as in ordinary object-oriented languages. For

[1]Since TAL currently requires finite domains, it is necessary to specify upper and lower bounds on the `Real` domain as well as the desired precision. This is also true for the `Integer` domain which will be used in later examples. However, these limitations are not relevant to the modeling issues covered in this article.

example, **tank1**, **tank2** and **tank3** all have a volume, despite the fact that the latter two are declared as FLOWTANK objects.

Naturally, subclasses can also add new attributes. For example, the FLOWTANK class keeps track of the current flow of water in or out of the tank, which is modeled as a flow attribute:

attr FLOWTANK.flow : `Real`

Initializing Attributes

Although it would have been possible to introduce special syntax for object initialization, similar to constructors in standard object-oriented languages, this only appears to be natural in the case where complete information about all objects is available.

The TAL logics allow the use of incomplete information – for example, due to sensor accuracy, one might only know that the initial volume of water in a tank is less than 0.02. Therefore, we still use plain TAL-C observation statements to partially or completely initialize attributes at time 0.

obs $\forall tank.[0]\ tank.\mathsf{volume} \leq 0.02$
obs $[0]\ \mathbf{tank2}.\mathsf{flow} \hat{=} 0 \land \mathbf{tank3}.\mathsf{flow} \hat{=} 0.12$

4.3.3 Methods

In a classical object-oriented view, a method is a sequence of code that is procedurally executed when the method is invoked. In our approach, however, a method is a set of formulas that must be satisfied whenever the method is invoked. Methods can be invoked over intervals of time, and several methods can be invoked concurrently.

Three different kinds of methods are defined: Accessors (which query the state of an object), mutators (which are called in order to change the state of an object), and constraint methods (which are not explicitly invoked but are active at all timepoints).

Accessors

Accessors are used for querying the state of an object. This can be done simply by retrieving the current value of an attribute, or by performing arbitrarily complex calculations as long as these calculations can be expressed within the logic being used.

An accessor is modeled using a *return value fluent*, a dynamic (non-persistent, non-durational) fluent that takes on the desired return value at all timepoints. For example, a simple query-volume() method for a water tank can be modeled by introducing a dynamic fluent query-volume(TANK) : `Real` and adding the following domain constraint:

$\textbf{acc}\ [t]\ \mathit{tank}.\text{query-volume}() \mathrel{\hat{=}} \mathit{value}(t, \mathit{tank}.\text{volume})$

Although this type of accessor may not appear very useful at first glance, the intention is that the attributes of a class (such as volume in TANK) should be considered private within that class, and that external callers should only use the externally available interface, such as the query-volume accessor. Actually enforcing this intention would require additional support from the tools being used to reason about an object-oriented narrative.

A slightly more complex accessor might determine whether the tank is full, which is the case if its volume equals its maximum volume (maxvol). This is done by declaring a dynamic return value fluent query-full(TANK) : boolean and using the following domain constraint:

$\textbf{acc}\ [t]\ \mathit{tank}.\text{query-full}() \leftrightarrow \mathit{value}(t, \mathit{tank}.\text{volume}) = \mathit{value}(t, \mathit{tank}.\text{maxvol})$

Mutators

Mutators can be called to change the internal state of an object, and are modeled as dependency constraints triggered by boolean *invocation fluents*.

To define a mutator method with $n \geq 0$ arguments of sorts $\langle s_1, \dots, s_n \rangle$ in class CLASS, it is first necessary to define a durational invocation fluent method(CLASS, $s_1, \dots, s_n$) with default value false. The method implementation consists of a dependency constraint that is triggered for an object *obj* only when the invocation fluent *obj*.method($x_1, \dots, x_n$) is true. For example, a mutator set-volume(Real) can be defined in class TANK as follows:

$\textbf{per}\ \forall t, \mathit{tank}, v \in \text{Real}.\mathit{Dur}(t, \mathit{tank}.\text{set-volume}(v), \textbf{false})$
$\textbf{dep}\ \forall t, \mathit{tank}, v \in \text{Real}.[t]\ \mathit{tank}.\text{set-volume}(v) \rightarrow \textbf{Set}([t]\ \mathit{tank}.\text{volume} \mathrel{\hat{=}} v)$

Calling the method requires making the invocation fluent true for the appropriate arguments at the timepoint when the method should be invoked. As usual, this is done using the **Set** macro, and therefore a TAL dependency constraint is required. For example, the volume of **tank1** can be set to 4.5 at time 2 as follows:

$\textbf{dep}\ \textbf{Set}([2]\ \textbf{tank1}.\text{set-volume}(4.5) \mathrel{\hat{=}} \textbf{true})$

This is simplified further by defining $\textbf{Call}(\tau, f) \overset{\text{def}}{=} \textbf{Set}([\tau]f \mathrel{\hat{=}} \textbf{true})$:

$\textbf{dep}\ \textbf{Call}(2, \textbf{tank1}.\text{set-volume}(4.5))$

Constraint Methods

Constraint methods model behaviors that should always be active. Instead of being triggered by invocation fluents, constraint methods are active at all timepoints. In a sense, they could be viewed as mutators that are continuously invoked. This allows many common RAC constructions such as state constraints to be expressed while keeping an object-oriented viewpoint.

The fact that the volume of water in a FLOWTANK changes according to the flow of water can be encoded as follows:

dep $\textbf{Set}([t+1]\mathit{tank}.\mathsf{volume} \hat{=} \mathit{value}(t, \mathit{tank}.\mathsf{volume} + \mathit{tank}.\mathsf{flow}))$

This concludes the discussion of the most basic concepts in object-orientation: Classes, objects, attributes, and methods. The following section will show how to reify the class structure in order to model method overriding in TAL-C, while Section 4.5 demonstrates how some additional object-oriented concepts, such as abstract classes and final methods, can be modeled.

4.4 Inheritance and Overriding

Although the concepts presented in the previous section are sufficient for modeling many domains, it is still possible to improve the elaboration tolerance of the models considerably by introducing the object-oriented concept of *overriding*: Allowing a subclass to re-implement a method in order to refine or specialize it.

This requires a way of disabling a method implementation that is inherited from a superclass, which is facilitated by providing the logic formulas with some additional information about the class structure used in a domain model.

4.4.1 Reifying the Class Structure

Since the TAL logics have no built-in support for allowing logic formulas to inspect the class (sort) structure of a particular domain, it is necessary to reify this structure. This can of course be done mechanically, and support for this is built into current versions of VITAL (Kvarnström, 2005).

The class structure is reified by mechanically constructing a TAL value domain `classname` containing all class names, and declaring and initializing a persistent boolean fluent[2] subclass(`classname`, `classname`), where $\mathsf{subclass}(c_1, c_2)$ is true iff c_1 is a subclass of c_2. For the water tank example, the definitions would be equivalent to the following:

domain `classname` :elements { OBJECT, TANK, FLOWTANK }
feature subclass(`classname`, `classname`) :domain boolean
  per $\forall t, \mathit{classname}_1, \mathit{classname}_2 . t > 0 \rightarrow \mathit{Per}(t, \mathsf{subclass}(\mathit{classname}_1, \mathit{classname}_2))$

  obs $\forall c_1 \in \texttt{classname}, c_2 \in \texttt{classname}$
    $[0]\,\mathsf{subclass}(c_1, c_2) \leftrightarrow$
      $((c_1 = \text{FLOWTANK} \wedge c_2 = \text{OBJECT}) \vee$
      $(c_1 = \text{FLOWTANK} \wedge c_2 = \text{TANK}) \vee$
      $(c_1 = \text{TANK} \wedge c_2 = \text{OBJECT}))$

Note that since subclass is persistent, it is sufficient to provide a value at time 0. This value will automatically propagate to all timepoints.

[2]Although we do not intend to change subclass relations over time, TAL-C has no support for time-independent functions and therefore a fluent must be used.

In addition to this, it is sometimes necessary to be able to identify the exact type of a certain object. To this end, an attribute class of type `classname` is added to the root class OBJECT:

attr OBJECT.class : `classname`

This attribute is also initialized automatically during the translation of the object declaration statements. In the water tank example, the following observations would be generated:

obs [0] **tank1**.class $\hat{=}$ TANK
obs [0] **tank2**.class $\hat{=}$ FLOWTANK
obs [0] **tank3**.class $\hat{=}$ FLOWTANK
obs [0] **pipe**.class $\hat{=}$ PIPE

It should be emphasized that these domains and fluents are created automatically during the translation process and need not be explicitly defined by the user.

4.4.2 Overriding Method Implementations

Suppose that a method method is defined and implemented in a class CLASS $\in$ `classname`. This implementation of method will be active for any object of type CLASS, including objects belonging to subclasses of CLASS.

When a new subclass SUB is created, we may want to override some of the methods defined in the superclass CLASS. This means not only adding a new implementation of the method for objects in SUB, but also *disabling* the old implementation for those objects.

To allow this to be modeled in TAL-C it is necessary to reify the concept of overriding a method. We introduce the boolean fluent override(SUB, method, CLASS) expressing the fact that for objects belonging to SUB, any implementation of method in the superclass CLASS is overridden and should be disabled. This fluent is durational with default value false, since overriding should only occur when explicitly forced.

All method implementations should then be conditionalized on not being overridden, and should explicitly override implementations in superclasses.

The former is achieved by adding a suitable override expression in the precondition of each method. For example, when set-volume mutator declared in class TANK is called for an object *tank*, the exact type of that object is *tank*.class (which may be TANK or FLOWTANK). Thus, the method should be disabled if for objects of this type (*tank*.class), the implementation of set-volume in the class TANK is overridden – in other words, if override(*tank*.class, set-volume, TANK). The method is therefore conditionalized as follows:

dep $\forall t, tank \in$ TANK, $f \in$ `Real`
    $[t]$ *tank*.set-volume$(f) \land \neg$override(*tank*.class, set-volume, TANK) $\rightarrow$
    Set$([t]$ *tank*.volume $\hat{=} f)$

The latter is done by adding a statement on the following form each time a method method is defined in a class CURRENTCLASS:

$\textbf{dep}\ \forall t, \textsc{super} \in \texttt{classname}, \textsc{sub} \in \texttt{classname}$
$\quad ([t]\ \mathsf{subclass}(\textsc{currentclass}, \textsc{super})) \land$
$\quad ([t]\ \mathsf{subclass}(\textsc{sub}, \textsc{currentclass}) \lor \textsc{sub} = \textsc{currentclass}) \rightarrow$
$\quad \textbf{Set}([t]\ \mathsf{override}(\textsc{sub}, \mathsf{method}, \textsc{super}))$

This states that when a method is re-implemented in CURRENTCLASS, its inherited implementation from any superclass SUPER is disabled for any object whose type SUB is either exactly CURRENTCLASS or a subclass of CURRENTCLASS.

For convenience, the macro **DisableInherited**(CURRENTCLASS, method) will be used as a shorthand for statements of this type. This yields the following final definition of the set-volume mutator:

$\textbf{dep}\ \textbf{DisableInherited}(\textsc{tank}, \mathsf{set\text{-}volume})$
$\textbf{dep}\ \forall t, \mathit{tank} \in \textsc{tank}, v \in \texttt{Real}$
$\quad [t]\ \mathit{tank}.\mathsf{set\text{-}volume}(v) \land \neg\mathsf{override}(\mathit{tank}.\mathsf{class}, \mathsf{set\text{-}volume}, \textsc{tank}) \rightarrow$
$\quad \textbf{Set}([t]\ \mathit{tank}.\mathsf{volume} \mathrel{\hat=} v)$

4.5 Additional Object-Oriented Concepts

This section will briefly present some additional ideas regarding the use of object-oriented modeling in a logic for reasoning about action and change. These ideas build on the basic concepts presented in the previous two sections, but will not be developed at the same level of detail. Rather, they are intended to demonstrate the flexibility of the paradigm and show how it could be extended and modified in various directions depending on the needs of the user.

4.5.1 Multiple Method Implementations

In the examples presented previously, a method always has a single implementation. However, there is no reason why this always has to be the case. For example, a mutator could consist of multiple dependency constraints, all of which are triggered by the same invocation fluent. This allows a more modular implementation of complex methods. It also permits a subclass to *add* to the implementation of a method, rather than replace it, simply by not calling the **DisableInherited** macro to disable the implementation provided by the superclass. This resembles the ability to call a superclass implementation of a method using super.method(...) in the Java programming language.

4.5.2 Preventing Overriding: Final Methods

In some object-oriented programming languages, a method implementation can be marked as "final", meaning that it cannot be overridden in a subclass.

Final methods can be defined in TAL-C by stating that they are never overridden. For example, the set-volume method from Section 4.4.2 could be made final by adding the following statement:

$\textsf{\textbf{acc}}\ \forall t, tank \in \text{TANK}.[t]\ \neg\textsf{override}(tank.\textsf{class}, \textsf{set-volume}, \text{TANK})$

Unlike most programming languages, this form of type checking is dynamic rather than static. If a method is overridden despite being final, this will generate an inconsistent narrative rather than an error during translation. VITAL will detect such inconsistencies and report the error to the user.

4.5.3 Forcing Overriding: Abstract Methods

While final methods are implemented and cannot be overridden in subclasses, abstract methods are *not* implemented and *must* be overridden in all subclasses. The following statement can be used to declare that the get-color method is abstract in the class TANK:

$\textsf{\textbf{acc}}\ \forall t, tank \in \text{TANK}.[t]\ \textsf{override}(tank.\textsf{class}, \textsf{get-color}, \text{TANK})$

Note that this statement in itself is not sufficient for permitting the override fluent to be true. The fluent is durational, and can only take on the value **true** if it is explicitly *assigned* that value, which is not the case in this formula. Instead, the formula states that someone else must have assigned it the value true using the **Set** macro, which would be done indirectly by an overriding method declaration using the **DisableInherited** macro.

4.5.4 Abstract Classes

An abstract class cannot be instantiated. Such a class can be modeled using a simple constraint of the following form:

$\textsf{\textbf{acc}}\ \forall t \neg \exists object.[t]\ object.\textsf{class} \hat{=} \text{CLASS}$

4.5.5 Class Methods

All methods shown up to now have been instance methods. For example, set-volume is called for an instance of the TANK class, and only alters the volume of that specific instance. It is also possible to model class methods, which are associated with the class itself rather than with an instance.

A class accessor method can be defined in TAL-C using a return value fluent that does not take an object as its first argument. Similarly, a class mutator can be defined using an invocation fluent that does not take an object as its first argument. For example, all tank volumes can be reset to zero using the following class method in the TANK class:

$$\textsf{\textbf{dep}}\ \forall t.[t]\ \textsf{set-zero-volume}() \land \neg\textsf{override}(\text{TANK}, \textsf{set-zero-volume}, \text{TANK}) \rightarrow \\ \forall tank.\textbf{Set}([t]\ tank.\textsf{volume} \hat{=} 0.0)$$

Note that this method is called directly, as in $\textbf{Call}(7, \textsf{set-zero-volume}())$, without specifying a tank object as in $\textbf{Call}(7, \textbf{tank1}.\textsf{set-volume}(0))$.

4.5.6 Access Control

For mutators, a form of cooperative access control can be implemented by adding to the invocation fluent another argument representing the caller. Using the set-volume mutator as an example, the following changes would be made:

$$\begin{array}{l}
\textbf{dep}\ \textbf{DisableInherited}(\text{TANK}, \mathsf{set\text{-}volume}) \\
\textbf{dep}\ \forall t, tank \in \text{TANK}, caller \in \text{TANK}, v \in \texttt{Real} \\
\quad [t]\ tank.\mathsf{set\text{-}volume}(caller, v) \land \neg\mathsf{override}(tank.\mathsf{class}, \mathsf{set\text{-}volume}, \text{TANK}) \rightarrow \\
\quad \textbf{Set}([t]\ tank.\mathsf{volume} \mathrel{\hat{=}} v)
\end{array}$$

In this definition, the caller argument must be a TANK, and consequently only a TANK can call the set-volume method. This is similar to a protected method in Java, and could possibly be used to help ensure that encapsulation is respected. However, this only provides a purely cooperative form of access control, since anyone wanting to call set-volume() could in principle simply send an arbitrary tank object as the caller.

4.6 Elaboration Tolerance through Object-Orientation

According to McCarthy (1998), elaboration tolerance is "the ability to accept changes to a person's or a computer program's representation of facts about a subject without having to start all over". Several ideas used in the object-oriented paradigm facilitate the creation of elaboration tolerant domain models. This is not surprising, since the reasons behind the object-oriented paradigm include modularization and the possibility to reuse code.

The structuring of objects, fluents, domain constraints and dependency constraints into a well-defined set of named classes, attributes and methods is a powerful tool for increasing the readability of a domain definition. This helps provide a better understanding of the domain, which is in itself a very important prerequisite for being able to adapt and extend the definition.

The use of inheritance makes it possible to specialize a class, adding new attributes, methods and constraints while reusing those features from the superclass that are still useful in the new subclass. Using overriding, the behaviors of a superclass can be changed without knowing implementation-specific details and without the need for "surgery" (McCarthy's term for modifying a domain description by actually changing or removing formulas or terms rather than merely adding facts).

While the creation of a subclass does not alter the behavior of its superclass, it is also possible to add new attributes and methods directly to an existing class without the need to modify the existing parts of the class definition.

Adding a new class requires changes to the `classname` domain and the subclass fluent. These changes are done automatically at translation time. Adding new methods may also yield a new definition of the automatically generated *Occlude* predicate (the TAL approach to solving the frame problem, as described in Appendices A and B). However, the new definition can be created by analyzing the new methods in isolation and adding new disjuncts to the existing definition of *Occlude*. It is not necessary to start over from the beginning because a new class is added or because a method is overridden.

The elaboration tolerance of this approach will now be tested using a concrete example domain.

4.7 Missionaries and Cannibals

McCarthy (1998) illustrates his ideas regarding elaboration tolerance with 19 elaborations of the Missionaries and Cannibals Problem (MCP). We will begin by modeling the basic, unelaborated domain using the object-oriented constructions presented above. In the next section we will show that the ability to override methods and to add new methods and attributes in subclasses provides a natural way to model many of the elaborations. Section 4.9 shows how the problem instances can be solved by generating plans within the logic.

4.7.1 Overview of the Design

The basic version of the MCP is as follows:

> Three missionaries and three cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How shall they cross in order to avoid anyone being eaten?

Although we know we will eventually need to model some elaborated versions of the domain, we will attempt to ignore that knowledge and provide a model suitable for this particular version of the MCP. This will provide a better test for whether the object-oriented model is truly elaboration tolerant.

We will define classes for objects, boats, places, and banks (Figure 4.1). Like Lifschitz (2000), we will initially model missionaries and cannibals as *groups* of a certain size rather than as individuals, despite the fact that a few of the elaborations do require individuals to be treated as such; this is also done to provide a better test for elaboration tolerance. In the standard domain, there will be six (possibly empty) groups: Missionaries and cannibals at the left bank, at the right bank, and on the boat.

4.7.2 Object

The root class OBJECT has a pos attribute representing its position, which is a PLACE (Section 4.7.3):

```
class OBJECT
 attr OBJECT.pos : PLACE
```

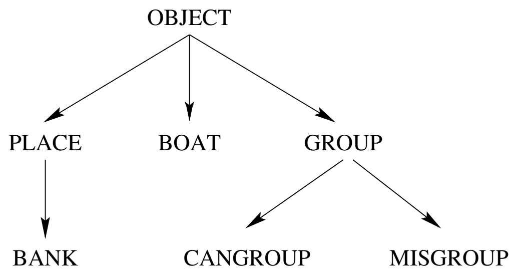


Figure 4.1: Classes in the Missionaries and Cannibals Domain

The following methods are available for accessing and changing the position:

Accessor query-pos(): Returns the position of the object.

dep **DisableInherited**(OBJECT, query-pos)
dep $[t]$ ¬override(*object*.class, query-pos, OBJECT) $\rightarrow$
  Set($[t]$ *object*.query-pos() $\hat{=}$ *value*(t, *object*.pos))

Mutator set-pos(PLACE): Sets the position of the object.

dep **DisableInherited**(OBJECT, set-pos)
dep $[t]$ ¬override(*object*.class, set-pos, OBJECT) $\wedge$ *object*.set-pos(*place*) $\rightarrow$
  Set($[t]$ *object*.pos $\hat{=}$ *place*)

In the remainder of this article, attributes will generally be assumed to have accessors and mutators following this pattern.

4.7.3 Place

The standard problem contains three different places: The left and right river bank and onboard the boat. This is modeled as a generic class PLACE with a subclass BANK.

A PLACE may be connected to other places, which is represented using a boolean attribute connection with a PLACE argument.

class PLACE *extends* OBJECT
attr PLACE.connection(PLACE) : `boolean`

Since the PLACE onboard the boat will be connected to the bank where it is currently located, and since the boat will move between the two banks, the connection attribute will change dynamically over time. Therefore two mutator methods are available, in addition to the standard query method.

Accessor query-connection(PLACE): Returns true if this PLACE is connected to the given PLACE.

```
dep DisableInherited(PLACE, query-connection)
dep [t] ¬override(place.class, query-connection, PLACE) →
    Set([t] place.query-connection(place′) ≙ value(t, place.connection(place′)))
```

Mutator add-connection(PLACE): Connects this PLACE to another PLACE.

```
dep DisableInherited(PLACE, add-connection)
dep [t] ¬override(place.class, add-connection, OBJECT) ∧
        place.add-connection(place′) →
    Set([t] place.connection(place′) ≙ true) ∧
    Set([t] place′.connection(place) ≙ true)
```

Mutator remove-connection(PLACE): Removes the connection from this PLACE to another PLACE.

```
dep DisableInherited(PLACE, remove-connection)
dep [t] ¬override(place.class, remove-connection, OBJECT) ∧
        place.remove-connection(place′) →
    Set([t] place.connection(place′) ≙ false) ∧
    Set([t] place′.connection(place) ≙ false)
```

4.7.4 Bank

A BANK is a PLACE where a boat can be located. The standard MCP has two banks: The left bank and the right bank.

```
class BANK extends PLACE
```

This class adds no new methods or attributes. Instead, the constraints on a BOAT will guarantee that it is always located at a BANK.

4.7.5 Group

A GROUP represents a group of people in a certain location; subclasses such as CANGROUP and MISGROUP will be used for specific types of people. It adds two new methods and a size attribute specifying the number of people in the group.

```
class GROUP extends OBJECT
 attr GROUP.size : Integer
```

Accessor query-can-move-to(GROUP): In the basic domain, people can move from one group to another only if they are groups of the same type and the two groups are connected. For example, people cannot move from a missionary group to a cannibal group, or teleport from the left bank to the right bank. For simplicity, we make the return value fluent durational with default value false, and explicitly set it to true only when necessary.

$$\begin{array}{l}
\textbf{dep DisableInherited}(\text{GROUP}, \mathsf{query\text{-}can\text{-}move\text{-}to}) \\
\textbf{dep } [t]\ \neg\mathsf{override}(\mathit{group}.\mathsf{class}, \mathsf{query\text{-}can\text{-}move\text{-}to}, \text{GROUP}) \land \\
\qquad \mathit{group}.\mathsf{query\text{-}pos}().\mathsf{query\text{-}connection}(\mathit{group}'.\mathsf{query\text{-}pos}()) \land \\
\qquad \mathit{group}.\mathsf{class} \hat{=} \mathit{group}'.\mathsf{class}) \rightarrow \\
\quad \textbf{Set}([t]\mathit{group}.\mathsf{query\text{-}can\text{-}move\text{-}to}(\mathit{group}') \hat{=} \textbf{true})
\end{array}$$

Mutator modify-group(GROUP, n): Calling $\mathit{group}.\mathsf{modify\text{-}group}(\mathit{group}_2, n)$ moves n people from *group* to group_2, if n is positive – otherwise, it moves $|n|$ people in the other direction. It is the caller's responsibility to use query-can-move-to() to ensure that the change is in fact "legal", and to ensure that a sufficient number of people is available in the source group. It is also the caller's responsibility to ensure that symmetry is retained: If $\mathit{group}.\mathsf{modify\text{-}group}(\mathit{group}_2, n)$ is called, the corresponding method $\mathit{group}_2.\mathsf{modify\text{-}group}(\mathit{group}, -n)$ must also be called.

The implementation of this method is somewhat complex due to the fact that people could potentially move concurrently between multiple groups. For example, one person could move from **group1** to **group2** while another moves from **group2** to **group3** and two from **group3** to **group1**. The cumulative effects of these concurrent method calls must be taken into account.

For this reason, modify-group does not follow the standard pattern where each invocation triggers a separate instance of a formula. Instead, a single dependency constraint sums the arguments of all concurrent invocations:[3]

$$\begin{array}{l}
\textbf{dep DisableInherited}(\text{GROUP}, \mathsf{modify\text{-}group}) \\
\textbf{dep } [t]\ \neg\mathsf{override}(\mathit{group}.\mathsf{class}, \mathsf{modify\text{-}group}, \text{GROUP}) \rightarrow \\
\quad \textbf{Set}([t+1]\,\mathit{group}.\mathsf{size} \hat{=} \mathit{value}(t, \mathit{group}.\mathsf{size}) + \displaystyle\sum_{\{\langle g,x\rangle \mid g\in\text{GROUP}\land[t]\,\mathit{group}.\mathsf{modify\text{-}group}(g,x)\}} x
\end{array}$$

The macro **people_at**$(\tau, \text{GROUP}, \mathit{place})$ will denote the number of people at *place* of the given type GROUP at time τ:

$$\textbf{people\_at}(\tau, \text{GROUP}, \mathit{place}) = \sum_{\{g \mid g\in\text{GROUP}\land[\tau]\, g.\mathsf{query\text{-}pos}()\hat{=}\mathit{place}\}} \mathit{value}(\tau, g.\mathsf{query\text{-}size}())$$

For example, given that **left** denotes the left bank, the macro expression **people_at** (7, CANGROUP, **left**) denotes the number of cannibals on the left bank at time 7, and **people_at**(7, GROUP, **left**) denotes the total number of people on the left bank at time 7.

[3]Throughout this article we will use summation over a set as a shorthand. Since TAL-C uses finite domains, each expression can be rewritten as a finite expression using plain addition.

4.7.6 Cannibals

A CANGROUP is a group of cannibals. The class extends GROUP and adds one new method.

class CANGROUP *extends* GROUP

Constraint eat-missionaries(): Specifies that there cannot be more cannibals than missionaries at any place. This constraint rules out any state where the cannibals would be able to eat a missionary.

Note that whenever a boat is at a river bank, anyone in the boat is considered to be at the same place as anyone on the bank. For this reason we define the macro **people-in-boats-near**(τ, GROUP, *place*), denoting the number of people in boats at the given place *place*, belonging to a group of the given type GROUP, at time τ:

$$\textbf{people-in-boats-near}(\tau, \text{GROUP}, \mathit{place}) = \sum_{\{\langle boat,g\rangle \mid [\tau]\, g.\text{query-pos}() \hat{=} boat.\text{query-onboard}() \wedge boat.\text{query-pos}() \hat{=} place\}} value(\tau, g.\text{query-size}())$$

Then, if *totalmis* is the total number of missionaries in a certain location (or in boats at that location), then either this must be zero or it must be greater than the total number of cannibals.

```
dep DisableInherited(CANGROUP, eat-missionaries)
acc [t] ¬override(cangroup.class, eat-missionaries, CANGROUP) ∧
        cangroup.query-position() ≙ place ∧
    totalmis = people_at(t, MISGROUP, place) +
               people-in-boats-near(t, MISGROUP, place) →
    totalmis = 0 ∨
    totalmis >= people_at(t, CANGROUP, place) +
                people-in-boats-near(t, CANGROUP, place)
```

4.7.7 Missionaries

A MISGROUP is a group of missionaries. The class extends GROUP and adds no new methods or attributes.

class MISGROUP *extends* GROUP

4.7.8 Boat

A BOAT is used to cross the river. Its onboard attribute points to the PLACE onboard the boat (which is the pos of any GROUP onboard the boat).

```
  class BOAT extends OBJECT
   attr BOAT.onboard : PLACE
```

There are two new methods:

Constraint boat-limit(): There must never be more than two passengers onboard a boat.

```
  dep DisableInherited(BOAT, boat-limit)
  dep [t] ¬override(boat.class, boat-limit, BOAT) →
        people_at(t, GROUP, value(t, boat.query-onboard())) ≤ 2
```

Mutator move-to(BANK): The move-to method is a low-level mutator that moves the boat to another BANK. This involves altering the pos attribute, but also removing the connection from the boat to its current location as well as adding a new connection from the boat to its new location.

```
  dep DisableInherited(BOAT, move-to)
  dep [t] ¬override(boat.class, move-to, BOAT) ∧
          boat.move-to(bank) ∧
          boat.query-pos() = oldbank →
        Call(t + 1, boat.query-onboard().remove-connection(oldbank)) ∧
        Call(t + 1, boat.set-pos(bank)) ∧
        Call(t + 1, boat.query-onboard().add-connection(bank))
```

4.7.9 Setting Up the Problem Instance

In order to set up a problem instance, we first have to instantiate some objects. The boat will be called **vera**, there will be two banks (**left** and **right**), and there are groups of missionaries and cannibals in all three places.

```
  obj left, right : BANK
  obj onvera : PLACE
  obj vera : BOAT
  obj cleft, cvera, cright : CANGROUP
  obj mleft, mvera, mright : MISGROUP
```

The following observation statements specify the attributes of these objects:

```
  obs [0] vera.pos ≙ left ∧ vera.onboard ≙ onvera
  obs [0] cleft.pos ≙ left ∧ cleft.size ≙ 3
  obs [0] cvera.pos ≙ onvera
  obs [0] cright.pos ≙ right
  obs [0] mleft.pos ≙ left ∧ mleft.size ≙ 3
  obs [0] mvera.pos ≙ onvera
  obs [0] mright.pos ≙ right
  acc [0] group.size ≙ 0 ↔ (group ≠ mleft ∧ group ≠ cleft)
```

$$\textsf{acc}\ [0]\ \mathit{place}_1.\textsf{connection}(\mathit{place}_2) \leftrightarrow \\ \quad ((\mathit{place}_1 = \textbf{left} \wedge \mathit{place}_2 = \textbf{onvera}) \vee \\ \quad (\mathit{place}_1 = \textbf{onvera} \wedge \mathit{place}_2 = \textbf{left}))$$

This completes the modeling of the basic Missionaries and Cannibals Domain. In the next section we will describe 19 elaborations of this domain, and in Section 4.9, we will show how to solve the problems within the logic.

4.8 Elaborations of the MCP Domain

McCarthy (1998) considers 19 different elaborations of the basic Missionaries and Cannibals domain, and discusses the requirements these domains place on a formalism used for modeling them and on a system for reasoning about and solving the problems. These elaborations will now be modeled in TAL-C using the object-oriented model of the MCP domain as a basis. The relations between the elaborations are shown in Figure 4.2.

The elaborations are often rather vaguely formulated, and we do not claim to have captured every aspect of each problem or that the formalism always allows the elaborations to be expressed as succinctly as possible. We concentrate on the modeling of the domains rather than on the computational properties of a reasoner finding plans for problem instances or proving that no plan exists. However, we do feel that most of the main points of the domain elaborations have been modeled in a reasonable manner.

Earlier versions of the domain definitions are available as part of the VITAL tool, which can be downloaded from the web (Kvarnström, 2005). The current versions will be added in the next release of VITAL.

4.8.1 Domain and Problem Specifications

We will consider each problem to consist of two parts. The *domain specification* defines the classes being used together with their attributes and the inheritance hierarchy, while the *problem specification* defines the object instances being used in a specific problem instance together with the initial values of their attributes.

Our focus has been on elaboration tolerance for the domain specification. Each elaboration may add new classes, or add new methods or attributes to existing classes. Note that no part of the original $\mathcal{L}(\text{ND})$ domain specification is removed or modified in any of the elaborations.

Although it would have been possible to use similar techniques to model the problem specification in Section 4.7.9 in a defeasible manner, we instead make the assumption that one is generally interested in solving many different problems in the same general domain and that the specific problem instances (such as the number of missionaries and cannibals, the set of river banks, and which places are con-

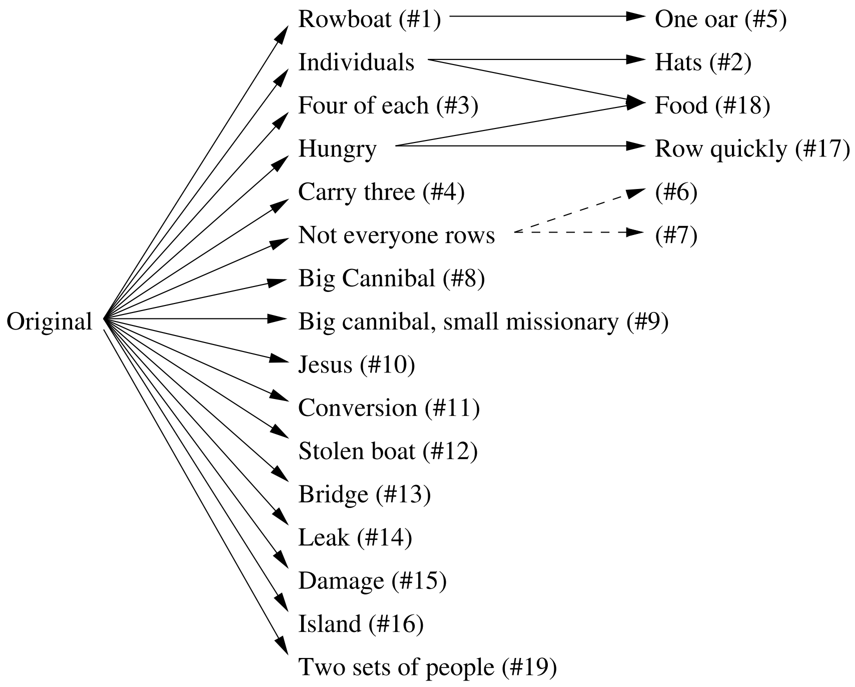


Figure 4.2: Elaborations of the Missionaries and Cannibals domain

nected) are generated from scratch each time. The problem instance definitions for the elaborations below are generally trivial and will usually be omitted.

4.8.2 The Boat Is a Rowboat (#1)

In the first elaboration by McCarthy, we find out that the boat is in fact a rowboat. This requires a new class ROWBOAT, subclass of BOAT, and **vera** must be made an instance of ROWBOAT.

However, no new information is given regarding rowboats. The elaborated scenario is essentially similar to the original problem – with the important exception that if further information about rowboats is presented in the future, we will be able to draw additional or different conclusions about **vera**.

```
class ROWBOAT extends BOAT
  obj vera : ROWBOAT
```

4.8.3 Missionaries and Cannibals Have Hats (#2)

In the second elaboration, the missionaries and cannibals have hats, all different. The hats may be exchanged among the missionaries and cannibals.

Viewing Missionaries and Cannibals as Individuals

While missionaries and cannibals used to be interchangeable and could be modeled as groups, they must now be seen as individuals. A class for persons is added, together with a group attribute that keeps track of the group to which the person belongs. This attribute should be initialized to suitable values in the initial state.

$$\begin{array}{l} \textbf{class}\ \text{PERSON}\ \textit{extends}\ \text{OBJECT} \\ \textbf{attr}\ \text{PERSON.group} : \text{GROUP} \end{array}$$

What remains is ensuring that a person always belongs to the right group. The only method moving people between groups is GROUP.modify-group(), but this method only specifies how many people should move to another group, not *which* people should move. Adapting this method to a model containing individuals may seem to be a quite complicated task, and it might even seem like this elaboration is beyond the capabilities of our logic. Fortunately, this is not the case.

The solution lies in making the group attribute dynamic – allowing it to vary freely over time without a persistence assumption – and then constraining it using a new addition to modify-group(). The additional constraint essentially states that if n people should move from $group_1$ to $group_2$, then there should be exactly n individuals who previously belonged to $group_1$ and now instead belong to $group_2$. Note that we do not *override* modify-group: We merely add to its previous definition.

Constraint modify-group(GROUP$_2$, n): Suppose that at some timepoint, the method $group_1$.modify-group$(group_2, n)$ is invoked, where $group_1$ and $group_2$ are two different groups.

The definition of this method in the superclass (Section 4.7.5) states that if n is positive, then n people should move from $group_1$ to $group_2$. This means that exactly n individuals that used to belong to $group_1$ should now belong to $group_2$. This is achieved using the first method implementation below.

On the other hand, if n is negative, then $-n$ people should move in the other direction. But in this case, $group_2$.modify-group$(group_1, -n)$ must also be called, according to the original constraints on modify-group in Section 4.7.5. Since $-n$ is positive, this case is also handled by the first method implementation below.

$$\begin{array}{ll} \textbf{acc} & [t]\ \neg\text{override}(group_1.\text{class}, \text{modify-group}, \text{GROUP}) \wedge \\ & [t+1]\ group_1.\text{modify-group}(group_2, n) \wedge \\ & n \geq 0 \wedge \\ & group_1 \neq group_2 \rightarrow \\ & \sum_{\{p \ \mid\ p \in \text{PERSON} \wedge [t]\, p.\text{group} \hat{=} group_1 \wedge [t+1]\, p.\text{group} \hat{=} group_2\}} 1 = \max(0, n) \end{array}$$

Yet another case occurs if for some timepoint t and some some distinct pair of groups $group_1$ and $group_2$, the method is not invoked at all (for any n). In this case, no person at all should move from $group_1$ to $group_2$. The rule above does not guarantee this, since if the method is not invoked at all for a certain pair of groups, the antecedent of the implication cannot hold. An additional method implementation is required, which is used when the method is *not* called:

```
**acc** $[t]\ \neg\mathsf{override}(group_1.\mathsf{class}, \mathsf{modify\text{-}group}, \textsc{Group}) \wedge$
    $[t+1]\ \neg\exists n[group_1.\mathsf{modify\text{-}group}(group_2, n)] \wedge$
    $group_1 \neq group_2 \rightarrow$
    $\sum_{\{p \mid p \in \textsc{Person} \wedge [t]\, p.\mathsf{group} \hat{=} group_1 \wedge [t+1]\, p.\mathsf{group} \hat{=} group_2\}} 1 = 0$
```

Note that the final line could also be written as follows:

$$\neg\exists person[[t]\ person.\mathsf{group} \hat{=} group_1 \wedge [t+1]\ person.\mathsf{group} \hat{=} group_2]$$

These two method implementations are sufficient to extend the group model into a model with individuals, together with a new problem instance definition where six PERSON objects are declared and placed into the groups on the left bank. This hybrid group/individual model is admittedly somewhat more complex than a pure individual-based model, but it is nevertheless interesting to see that the model can be adjusted in this way without having to remove or completely rewrite existing classes and methods.

It should be noted that this implementation makes it impossible to move $n \geq 0$ people from *group* to $group_2$ and at the same time move $n' \geq 0$ people from *group* to $group_2$, where $n \neq n'$. Although one could possibly interpret this to mean that $n + n'$ people move from *group* to $group_2$, this would only introduce complications that are generally unnecessary.

Hats

Given the domain presented above, where the missionaries and cannibals are seen as individuals, adding hats and the possibility to exchange them is trivial. A new class for hats is added, together with a new hat attribute for determining which hat belongs to which person:

```
**class** HAT *extends* OBJECT
 **attr** PERSON.hat : HAT
```

Accessor and mutator methods for the hat attribute are added. Also, a method for exchanging hats is added to PERSON:

Mutator exchange-hats(PERSON): Exchange hats with the given person.

```
**dep** **DisableInherited**(PERSON, exchange-hats)
**dep** $[t]\ \neg\mathsf{override}(person.\mathsf{class}, \mathsf{exchange\text{-}hats}, \textsc{Person}) \wedge$
        $person.\mathsf{exchange\text{-}hats}(person') \rightarrow$
    $\mathbf{Call}(t+1, person.\mathsf{set\text{-}hat}(value(t, person'.\mathsf{get\text{-}hat}()))) \wedge$
    $\mathbf{Call}(t+1, person'.\mathsf{set\text{-}hat}(value(t, person.\mathsf{get\text{-}hat}())))$
```

Finally, six hats must be created and the hat attribute must be initialized.

4.8.4 Four of Each (#3)

There are four missionaries and four cannibals.

In our terminology, this is a change in the problem specification rather than in the domain specification. The problem specification is therefore modified accordingly:

obs $[0]$ **cleft**.pos $\hat{=}$ **left** $\wedge$ **cleft**.size $\hat{=} 4$
obs $[0]$ **mleft**.pos $\hat{=}$ **left** $\wedge$ **mleft**.size $\hat{=} 4$
obs $\ldots$

4.8.5 The Boat Can Carry Three (#4)

In the fourth elaboration, the boat can carry three people, while in the original MCP, the number of people onboard a BOAT was restricted to two. Although it was obvious that it would be useful to be able to model boats of varying capacities, we nonetheless deliberately chose to hardcode the capacity in the original boat-limit method in order to test the elaboration tolerance of the model. Thus, we now need to create a subclass that overrides the old constraint. But this time, it will be done the right way:

class SIZEBOAT *extends* BOAT
attr SIZEBOAT.capacity : `Integer`

Constraint boat-limit(): Ensure that the capacity is not exceeded.

dep **DisableInherited**(SIZEBOAT, boat-limit)
acc $[t]$ $\neg$override(*sizeboat*.class, boat-limit, SIZEBOAT) $\rightarrow$
    people_at(t, GROUP, *value*(t, *sizeboat*.query-onboard())) $\leq$
    value(t, *sizeboat*.query-capacity())

Using the capacity attribute it is now possible to model boats with arbitrary limits on the number of passengers.

4.8.6 One Oar on Each Bank (#5)

Suppose that the boat is a rowboat, and that there is initially one oar on each bank. Suppose also that one person can cross the river with a single oar, but that two people will need both oars to cross together.

Modeling this as an extension of elaboration 1 requires a new class for oars, and two oars must be created and placed in their initial positions. These oars can later be moved between connected positions using set-pos().

class OAR *extends* OBJECT
obj **oar1**, **oar2** : OAR
obs $[0]$ **oar1**.pos $\hat{=}$ **left**
obs $[0]$ **oar2**.pos $\hat{=}$ **right**

It is also necessary to ensure that the boat only moves when a sufficient number of oars are available. One person can row using one oar, and two persons can row using two oars – in other words, the number of people in the boat must not exceed the number of oars.

$$
\begin{array}{l}
\textbf{dep}\ \textbf{DisableInherited}(\text{ROWBOAT}, \mathsf{oar\text{-}limit}) \\
\textbf{acc}\ [t]\ \neg\mathsf{override}(\mathit{rowboat}.\mathsf{class}, \mathsf{oar\text{-}limit}, \text{ROWBOAT}) \wedge \\
\qquad \mathit{rowboat}.\mathsf{query\text{-}onboard}() \mathrel{\hat{=}} \mathit{place} \rightarrow \\
\quad \textbf{people\_at}(t, \text{GROUP}, \mathit{place}) \leq \sum_{o \,\mid\, o \in \text{OAR} \wedge [t]\, o.\mathsf{query\text{-}pos}() \hat{=} \mathit{place}} 1
\end{array}
$$

4.8.7 Not Everybody Can Row (#6 and #7)

In elaboration 6, only one cannibal and one missionary can row (which leaves the problem solvable), while in elaboration 7, no missionary can row (which makes it unsolvable). These elaborations extend elaboration 1 (the rowboat). Two new classes for rowing cannibals and rowing missionaries are introduced, and the problem initialization is changed accordingly (for example, six new groups are added):

$$
\begin{array}{l}
\textbf{class}\ \text{ROWCANGROUP}\ \mathit{extends}\ \text{CANGROUP} \\
\textbf{class}\ \text{ROWMISGROUP}\ \mathit{extends}\ \text{MISGROUP} \\
\quad \textbf{obj}\ \textbf{rcleft}, \textbf{rcvera}, \textbf{rcright} : \text{ROWCANGROUP} \\
\quad \textbf{obj}\ \textbf{rmleft}, \textbf{rmvera}, \textbf{rmright} : \text{ROWMISGROUP} \\
\quad \textbf{obs}\ \ldots
\end{array}
$$

The new constraint method BOAT.row-limit() ensures that no boat moves unless there is someone aboard who can row.

$$
\begin{array}{l}
\textbf{dep}\ \textbf{DisableInherited}(\text{BOAT}, \mathsf{row\text{-}limit}) \\
\textbf{acc}\ [t]\ \neg\mathsf{override}(\mathit{boat}.\mathsf{class}, \mathsf{row\text{-}limit}, \text{BOAT}) \wedge \\
\qquad \mathit{boat}.\mathsf{query\text{-}pos}() \not\hat{=} \mathit{value}(t+1, \mathit{boat}.\mathsf{query\text{-}pos}()) \rightarrow \\
\quad \textbf{people\_at}(t, \text{ROWCANGROUP}, \mathit{boat}.\mathsf{query\text{-}onboard}()) + \\
\quad \textbf{people\_at}(t, \text{ROWMISGROUP}, \mathit{boat}.\mathsf{query\text{-}onboard}()) > 0
\end{array}
$$

4.8.8 Big Cannibal (#8)

In the eighth elaboration, one cannibal is too big to fit into the boat with another person. A new group class for big cannibals is introduced, and the problem specification is changed accordingly:

$$
\begin{array}{l}
\textbf{class}\ \text{BIGCANGROUP}\ \mathit{extends}\ \text{CANGROUP} \\
\quad \textbf{obj}\ \textbf{bcleft}, \textbf{bcvera}, \textbf{bcright} : \text{BIGCANGROUP} \\
\quad \textbf{obs}\ \ldots
\end{array}
$$

A new constraint method is added to this class, to ensure that if any big cannibals are on board a boat, then there is exactly one person on board that boat:

$$
\begin{array}{l}
\textbf{dep}\ \textbf{DisableInherited}(\text{BIGCANGROUP}, \mathsf{size\text{-}limit}) \\
\textbf{acc}\ [t]\ \neg\mathsf{override}(\mathit{bigcangroup}.\mathsf{class}, \mathsf{size\text{-}limit}, \text{BOAT}) \wedge \\
\quad \textbf{people\_at}(t, \text{BIGCANGROUP}, \mathit{boat}.\mathsf{query\text{-}onboard}()) > 0 \rightarrow \\
\quad \textbf{people\_at}(t, \text{GROUP}, \mathit{boat}.\mathsf{query\text{-}onboard}()) \mathrel{\hat{=}} 1
\end{array}
$$

4.8.9 Big Cannibal, Small Missionary (#9)

There is a big cannibal and a small missionary. The big cannibal can eat the small missionary if they are alone in the same place.

To model this elaboration, we add the classes SMALLMISGROUP for small missionaries and BIGCANGROUP for large cannibals together with a constraint method eat-small that ensures that a small missionary and a big cannibal are never isolated together.

```
class SMALLMISGROUP extends MISGROUP
class BIGCANGROUP extends CANGROUP
  dep DisableInherited(BIGCANGROUP, eat-small)
  acc [t] ¬override(bigcangroup.class, eat-small, BIGCANGROUP) ∧
      people_at(t, BIGCANGROUP, place) = 1 ∧
      people_at(t, SMALLMISGROUP, place) = 1 →
      people_at(t, GROUP, place) > 2
```

4.8.10 Jesus (#10)

One of the missionaries is Jesus Christ, who can walk on water. A new group class is created, and objects are instantiated and initialized for each position:

```
class JESUSGROUP extends MISGROUP
  obj jleft, jvera, jright : JESUSGROUP
  obs ...
```

The query-can-move-to() method from Section 4.7.5 is then overridden with a variation that does not require the origin and the destination to be connected.

Accessor query-can-move-to($\text{JESUSGROUP}'$): Jesus objects can move between non-connected places (that is, cross the river without a boat).

```
  dep DisableInherited(JESUSGROUP, query-can-move-to)
  dep [t] ¬override(jesusgroup.class, move-persons, JESUSGROUP) ∧
          jesusgroup.class ≙ jesusgroup′.class) →
      Set(jesusgroup.query-can-move-to(jesusgroup′) ≙ true)
```

4.8.11 Conversion (#11)

Three missionaries together can convert an isolated cannibal. Add a constraint method convert in class MISGROUP:

```
  dep DisableInherited(MISGROUP, convert)
  dep [t] ¬override(misgroup.class, convert, MISGROUP) ∧
      people_at(t, MISGROUP, place) ≥ 3 ∧
      people_at(t, CANGROUP, place) = 1 →
      Call(t + 1, misgroup.modify-group(misgroup, 1)) ∧
      Call(t + 1, misgroup.modify-group(cangroup, −1))
```

This elaboration takes advantage of the true concurrency in TAL-C (Karlsson & Gustafsson, 1999). For example, modify-group automatically handles the situation where a cannibal is boarding a boat while another is being converted to a missionary.

4.8.12 The Boat Might Be Stolen (#12)

Whenever a cannibal is alone in a boat, there is a 1/10 probability that he will steal it. Although TAL-C has no support for probability reasoning, it is possible to determine the probability that any particular boat will be stolen using an attribute prob-not-stolen initialized to 1.0. Whenever a cannibal is alone in a boat, the constraint method update-prob multiplies prob-not-stolen by 0.9; the value of *boat*.prob-not-stolen at the final timepoint of a model is the probability of that particular plan succeeding.

attr BOAT.prob-not-stolen : `Real`
obs $\forall boat.[0] boat.\text{prob-not-stolen} \mathrel{\hat{=}} 1.0$

dep $\textbf{DisableInherited}(\text{BOAT}, \text{update-prob})$
dep $[t]\ \neg\text{override}(boat.\text{class}, \text{update-prob}, \text{BOAT}) \wedge boat.\text{query-onboard}() \mathrel{\hat{=}} place \wedge$
$\textbf{people\_at}(t, \text{GROUP}, place) = 1 \wedge$
$\textbf{people\_at}(t, \text{CANGROUP}, place) = 1 \rightarrow$
$\textbf{Set}([t+1]\ boat.\text{prob-not-stolen} \mathrel{\hat{=}} 0.9 * value(t, boat.\text{prob-not-stolen}))$

4.8.13 The Bridge (#13)

There is a bridge. The capacity of the bridge is not specified, but as long as at least two people can cross simultaneously, an arbitrary number of people can cross. Add a BRIDGE class and ensure that its capacity limit is respected.

class BRIDGE *extends* PLACE
attr BRIDGE.capacity : `Integer`
dep $\textbf{DisableInherited}(\text{BRIDGE}, \text{bridge-limit})$
acc $[t]\ \neg\text{override}(bridge.\text{class}, \text{bridge-limit}, \text{BRIDGE}) \rightarrow \textbf{people\_at}(t, \text{GROUP}, bridge) \leq$
$value(t, bridge.\text{query-capacity}())$

Then instantiate a bridge, provide it with a capacity and connect it to the left and right banks.

4.8.14 The Boat Leaks (#14)

In elaboration 14, the boat leaks and must be bailed. Add a new durational boolean attribute bailed with default value false. The intention is that bailing the boat at a specific timepoint makes bailed true at that timepoint. A constraint method requires that the boat always be bailed (but does not cause the boat to be bailed – the user, or the controller, must call the bail method).

attr BOAT.bailed : boolean
dep **DisableInherited**(BOAT, bail)
dep $[t]$ ¬override(*boat*.class, bail, BOAT) $\rightarrow$ I($[t]$ *boat*.set-bailed(**true**))
dep **DisableInherited**(BOAT, must-bail)
acc $[t]$ ¬override(*boat*.class, must-bail, BOAT) $\rightarrow$ $[t]$ *boat*.query-bailed()

4.8.15 The Boat Can Be Damaged (#15)

The boat may suffer damage and have to be taken back to the left side for repairs. In this elaboration, the boat cannot move between banks instantaneously. We add a new bank **onriver** and a new class SLOWBOAT for boats that spend some time on the river before arriving at the destination. We also add a temporal constant *crosstime* representing the amount of time required to cross the river.

class SLOWBOAT *extends* BOAT
attr SLOWBOAT.emergency : BOOLEAN
obj **onriver** : BANK

The move-to method, which is responsible for moving the boat to another BANK, must also be overridden and split into two parts: (1) move the boat to **onriver**, and (2) after *crosstime* timepoints, if there has been no emergency, move it to the desired bank. The second part takes advantage of TAL-C's ability to handle delays (Doherty & Gustafsson, 1998; Karlsson et al., 1998).

Mutator move-to(BANK): Move the boat to another bank, with a delay.

dep **DisableInherited**(SLOWBOAT, move-to)
dep $[t]$ ¬override(*slowboat*.class, move-to, SLOWBOAT) $\wedge$
    slowboat.move-to(*bank*) $\wedge$
    slowboat.query-pos() $=$ *oldbank* $\rightarrow$
  Call($t+1$, *slowboat*.query-onboard().remove-connection(*oldbank*)) $\wedge$
  Call($t+1$, *slowboat*.set-pos(**onriver**))

dep $[t]$ ¬override(*slowboat*.class, move-to, SLOWBOAT) $\wedge$
    slowboat.move-to(*bank*) $\wedge$
  $[t+1, t+\mathit{crosstime}]$ ¬*slowboat*.query-emergency() $\rightarrow$
  Call($t+\mathit{crosstime}$, *slowboat*.set-pos(*bank*)) $\wedge$
  Call($t+\mathit{crosstime}$, *slowboat*.query-onboard().add-connection(*bank*))

If there is an emergency, the second dependency constraint above will not be triggered, and the boat will not end up at its intended destination. Instead, the boat should move to the left bank and be repaired.

Constraint emergency-behavior: If there are people on board and repairs are necessary, automatically move to the left bank for repairs.

dep **DisableInherited**(SLOWBOAT, emergency-behavior)
dep $[t]$ ¬override(*slowboat*.class, emergency-behavior, SLOWBOAT) ∧
slowboat.query-emergency() ∧
people_at(t, BOAT, *slowboat*.query-onboard()) $> 0 \rightarrow$
Call($t+1$, *slowboat*.set-pos(**left**)) ∧
Call($t+1$, *place*.add-connection(**left**)) ∧
Call($t+1$, *slowboat*.set-emergency(⊥))]

4.8.16 The Island (#16)

If an island is added, the problem can be solved with four missionaries and four cannibals. It is sufficient to change the number of people initially present on the left bank and add an island object:

obj **island** : BANK

4.8.17 Four Cannibals, Four Missionaries, Row Quickly (#17)

Elaboration 17 is defined as follows by McCarthy:

> There are four cannibals and four missionaries, but if the strongest of the missionaries rows fast enough, the cannibals won't have gotten so hungry that they will eat the missionaries. This could be made precise in various ways, but the information is usable even in vague form.

First, two new group classes are introduced: One for strong missionaries, and one for cannibals that may or may not be hungry. The necessary instances are created and initialized.

class HCANGROUP *extends* CANGROUP
class STMISGROUP *extends* MISGROUP
obj **hcleft**, **hcvera**, **hcright** : HCANGROUP
obj **smleft**, **smvera**, **smright** : STMISGROUP
obs ...

A new boolean attribute is introduced to keep track of whether the cannibals in a certain group are hungry or not. In the initial state, nobody is hungry.

attr HCANGROUP.hungry : `boolean`
obs $\forall$*hcangroup*.[0] *hcangroup*.hungry $\hat{=}$ **false**

The old eat-missionaries constraint stated unconditionally that the missionaries must never be outnumbered by the cannibals in any location. This constraint must be weakened slightly: If none of the cannibals at a certain location are hungry, it does not matter whether the missionaries are outnumbered or not.

```
dep DisableInherited(HCANGROUP, eat-missionaries)
acc [t] ¬override(hcangroup.class, eat-missionaries, HCANGROUP) ∧
        hcangroup.query-position() ≙ place ∧
        hcangroup.query-hungry() ∧
    totalmis = people_at(t, MISGROUP, place) +
               people-in-boats-near(t, MISGROUP, place) →
    totalmis = 0 ∨
    totalmis >= people_at(t, HCANGROUP, place) +
                people-in-boats-near(t, HCANGROUP, place)
```

What remains is determining exactly when the cannibals should become hungry. The information given by McCarthy could be interpreted in many different ways. It would be possible to model the strength of each person, let the amount of time required to cross the river depend on the strength of the rowers, and let every cannibal become hungry at, say, time 10. Although this could be modeled in TAL-C, we choose a simpler interpretation where the cannibals immediately become hungry when the strong missionary is no longer in the boat.

```
dep DisableInherited(HCANGROUP, become-hungry)
dep [t] ¬override(hcangroup.class, become-hungry, HCANGROUP) ∧
    t ≥ 1 ∧
    people_at(t, STMISGROUP, boat.query-onboard()) < 1 →
    Call(t + 1, hcangroup.set-hungry(true))
```

4.8.18 Four Cannibals, Four Missionaries, Food (#18)

Like in the previous elaboration, there are four missionaries and four cannibals, and the cannibals are initially not hungry. The difference is that in this elaboration, the missionaries have some food that they can give to the cannibals whenever they become hungrier. As McCarthy notes, this requires comparing a situation and a successor situation, which is clearly not a problem in TAL-C.

This is a quite complex elaboration. Since the level of hunger cannot be associated with a group, it requires treating people as individuals, and we will use elaboration 2 as the starting point. To this we will have to add a way of determining when to feed the cannibals, and keep track of how hungry they are and how much food each missionary has.

We begin by creating the subclasses FOODCANGROUP and FOODMISGROUP, in which some new methods will be added and others will be overridden. We also need the classes MISSIONARY and CANNIBAL, subclasses of PERSON (which was inherited from elaboration 2).

```
class FOODCANGROUP extends CANGROUP
class FOODMISGROUP extends MISGROUP
class MISSIONARY extends PERSON
class CANNIBAL extends PERSON
```

obj **cleft**, **cvera**, **cright** : FOODCANGROUP
obj **mleft**, **mvera**, **mright** : FOODMISGROUP
obj **misA**, **misB**, **misC**, **misD** : MISSIONARY
obj **canA**, **canB**, **canC**, **canD** : CANNIBAL

Cannibals can have different levels of hunger, modeled as an integer attribute. Missionaries have a certain amount of food. This must be initialized at time zero, and arbitrary numbers have been used below.

attr CANNIBAL.hunger : `Integer`
attr MISSIONARY.food : `Integer`
obs $[0]$ **canA**.hunger $\hat{=} 1 \wedge$ **canB**.hunger $\hat{=} 0 \wedge$
canC.hunger $\hat{=} 0 \wedge$ **canD**.hunger $\hat{=} 0$
obs $[0]$ **misA**.food $\hat{=} 3 \wedge$ **misB**.food $\hat{=} 1 \wedge$
misC.food $\hat{=} 7 \wedge$ **misD**.food $\hat{=} 7$

The feed method feeds a cannibal a certain amount of food. As in the modify-group method, two dependency constraints sum the arguments of all concurrent method invocations.

dep **DisableInherited**(MISSIONARY, feed)

dep $[t]$ $\neg$override($\mathit{missionary}$.class, feed, MISSIONARY) $\rightarrow$

$$\mathbf{Set}([t+1]\mathit{missionary}.\mathsf{food} \mathrel{\hat{=}} \mathit{value}(t, \mathit{missionary}.\mathsf{food}) - \sum_{\{\langle c,x\rangle \mid c \in \text{CANNIBAL} \wedge [t]\, \mathit{missionary}.\mathsf{feed}(c,x)\}} x$$

dep $[t]$ $\neg$override($\mathit{missionary}$.class, feed, MISSIONARY) $\rightarrow$

$$\mathbf{Set}([t+1]\mathit{cannibal}.\mathsf{hunger} \mathrel{\hat{=}} \mathit{value}(t, \mathit{cannibal}.\mathsf{hunger}) + \sum_{\{\langle m,x\rangle \mid m \in \text{MISSIONARY} \wedge [t]\, m.\mathsf{feed}(\mathit{cannibal},x)\}} x$$

If a cannibal is becoming hungrier, the missionaries may or may not feed him.

dep **DisableInherited**(MISSIONARY, do-feed)
dep $[t]$ $\neg$override($\mathit{missionary}$.class, do-feed, MISSIONARY) $\wedge$
$\mathit{missionary}$.query-group() $\hat{=}$ $\mathit{foodmisgroup} \wedge$
$\mathit{cannibal}$.query-group() $\hat{=}$ $\mathit{foodcangroup} \wedge$
$\mathit{foodmisgroup}$.query-pos() $\hat{=}$ $\mathit{foodcangroup}$.query-pos() $\wedge$
$[t+1]$ $\mathit{cannibal}$.query-hunger() $>$ $\mathit{value}(t, \mathit{cannibal}$.query-hunger()) $\rightarrow$
$\exists n. 0 \leq n \leq 1 \wedge$ **Call**($t+2, \mathit{missionary}$.feed($\mathit{cannibal}, n$))

The cannibals must become hungrier now and then. For example, they might become hungrier at time 2 and 4:

dep $t = 2 \vee t = 4 \rightarrow$ **Set**($[t+1]$ $\mathit{cannibal}$.hunger $\hat{=}$ $\mathit{value}(t, \mathit{cannibal}$.hunger$) + 1$

Finally, the original eat-missionaries constraint stated unconditionally that the missionaries must never be outnumbered by the cannibals in any location. Again, this constraint must be weakened slightly: If none of the cannibals at a certain location has a hunger level greater than 2, it does not matter whether the missionaries are outnumbered or not.

$$\begin{aligned}
&\textbf{dep DisableInherited}(\text{FOODCANGROUP}, \mathsf{eat\text{-}missionaries})\\
&\textbf{acc}\ [t]\ \neg\mathsf{override}(\mathit{foodcangroup}.\mathsf{class}, \mathsf{eat\text{-}missionaries}, \text{FOODCANGROUP}) \wedge\\
&\qquad \mathit{foodcangroup}.\mathsf{query\text{-}position}() \mathrel{\hat{=}} \mathit{place} \wedge\\
&\qquad (\exists \mathit{cannibal}.\mathit{cannibal}.\mathsf{get\text{-}group}() \mathrel{\hat{=}} \mathit{foodcangroup} \wedge\\
&\qquad\qquad\qquad \mathit{cannibal}.\mathsf{get\text{-}hunger}() > 2) \wedge\\
&\quad \mathit{totalmis} = \textbf{people\_at}(t, \text{FOODMISGROUP}, \mathit{place}) +\\
&\qquad\qquad \textbf{people-in-boats-near}(t, \text{FOODMISGROUP}, \mathit{place}) \rightarrow\\
&\quad \mathit{totalmis} = 0 \vee\\
&\quad \mathit{totalmis} >= \textbf{people\_at}(t, \text{FOODCANGROUP}, \mathit{place}) +\\
&\qquad\qquad \textbf{people-in-boats-near}(t, \text{FOODCANGROUP}, \mathit{place})
\end{aligned}$$

4.8.19 Two Sets of People (#19)

In the final elaboration, there are two sets of missionaries and cannibals too far apart along the river to interact. A new attribute same-set keeps track of which banks belong to the same "set", and must be initialized using observation statements:

$$\begin{aligned}
&\textbf{attr}\ \text{BANK}.\mathsf{same\text{-}set}(\text{BANK}) : \mathtt{boolean}\\
&\textbf{obs}\ [0]\ \textbf{left}.\mathsf{same\text{-}set}(\textbf{right}) \wedge \ldots
\end{aligned}$$

The following constraint method ensures that the origin and destination are in the same set.

$$\begin{aligned}
&\textbf{dep DisableInherited}(\text{BOAT}, \mathsf{move\text{-}same\text{-}set})\\
&\textbf{dep}\ [t]\ \neg\mathsf{override}(\mathit{boat}.\mathsf{class}, \mathsf{move\text{-}same\text{-}set}, \text{BOAT}) \rightarrow\\
&\qquad \mathit{boat}.\mathsf{query\text{-}pos}().\mathsf{query\text{-}same\text{-}set}(\mathit{value}(t+1, \mathit{boat}.\mathsf{query\text{-}pos}()))
\end{aligned}$$

4.8.20 Classes in the Elaborated Missionaries and Cannibals Problems

In the elaborations presented above we created a number of new classes that extend the class hierarchy shown in Figure 4.1. An overview of the new class hierarchy is shown in Figure 4.3.

4.9 Solving the Missionaries and Cannibals Problems

Though the main focus of this article is on modeling, we would also like to actually *solve* the Missionaries and Cannibals problem instances presented by McCarthy. In other words, given that the missionaries and cannibals are located on the left river bank, a suitable set of actions (or method invocations) should be found that moves everyone to the right bank without any missionaries being eaten.

Although one could use the model only for prediction and then apply standard planning algorithms to solve each problem, we instead choose to build on the ideas for automatic control presented in Gustafsson (2001) and model a controller within the logic. Since the different elaborations have slightly different demands on the

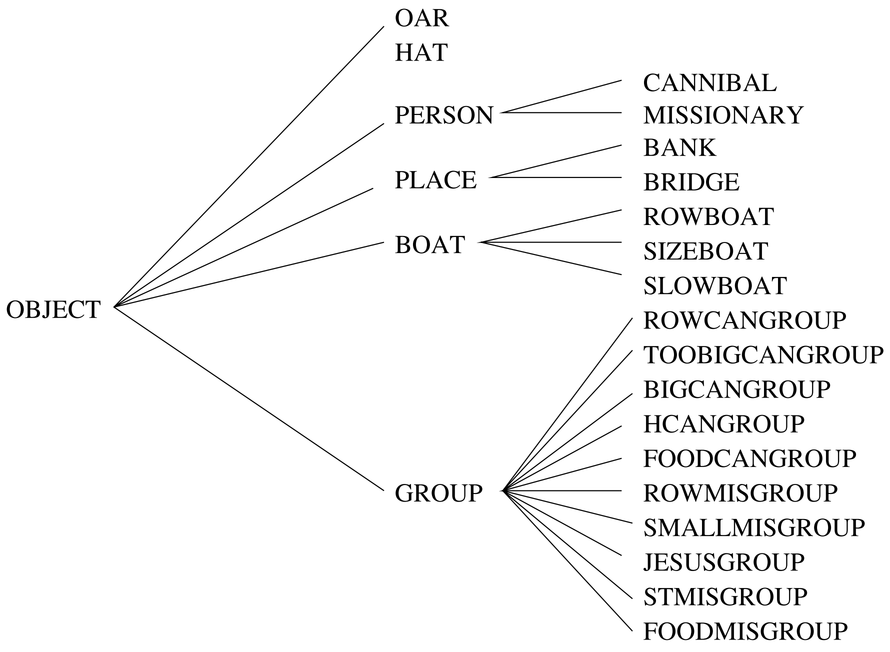


Figure 4.3: Classes in the Elaborated Missionaries and Cannibals Problems

controller, it will be modeled as another class whose methods can be overridden in subclasses, providing another test of the elaboration tolerance of the object-oriented approach.

The main idea behind the controller is that whenever there is a choice between different actions that could be invoked, this choice is modeled using an incompletely specified constraint method. For example, whenever a boat can move, a constraint method in the controller will call the boat's set-pos method to move it, but the exact destination will not be specified.

Every logical model of the resulting narrative corresponds to a different set of actions that could potentially be taken by the missionaries and cannibals, given that the cannibals never outnumber the missionaries in any location as required by eat-missionaries() (Section 4.7.6). What remains is choosing a model that actually achieves the goal, rather than just containing missionaries and cannibals moving around randomly. To achieve this, we assume (like Lifschitz, 2000) that we know the length t_* of the plan to be generated. By constraining the state at time t_* to be a solution state, where everyone is at the right river bank, we ensure that any remaining logical model must correspond to a valid plan.[4] The value t_* is made

[4]Note that this procedure depends on the fact that all incomplete information corresponds to possible choices of actions rather than incomplete knowledge about the world.

available in the narrative as a temporal constant, and will be used in some of the controller methods.

For the original problem, we know that the minimal plan length is 12. The plan lengths for the 19 elaborations will be shown together with the timing results in Section 4.9.3, and the goals must of course also be altered for those elaborations that involve different group types or a larger number of missionaries and cannibals.

obs $t_* = 12$
obs $[t_*]$ **mright**.size $\hat{=} 3 \wedge$ **cright**.size $\hat{=} 3$

4.9.1 A Controller for the Original Problem

The controller for the original problem will consist of a class CONTROLLER with a set of constraint methods defined below. One instance must be created in every elaboration.

class CONTROLLER *extends* OBJECT
  obj ctrl : CONTROLLER

Allowing People to Move

The first step in defining the controller is allowing people to move randomly between groups in connected locations. This is done by adding the following method:

Constraint move-persons(): Moves an unspecified number of people (possibly zero) between compatible groups in connected locations, where the compatibility is tested using the query-can-move-to method. For example, if there is a group of cannibals $group_1$ on the left bank and a group of cannibals $group_2$ on the boat, and the boat is at the left bank (the places are connected), then cannibals may move between $group_1$ and $group_2$. Note that GROUPs never move – people move by changing the size of two groups. Also note that the number of people moving from $group_1$ to $group_2$ can naturally be equal to zero.

The exact number of people moved by this method will be constrained indirectly by the goal as described above.

dep **DisableInherited**(CONTROLLER, move-persons)
dep $[t]$ $\neg$override($controller$.class, move-persons, CONTROLLER) $\wedge$
    $group_1$.query-can-move-to($group_2$) $\rightarrow$
  $\exists n\ [-value(t, group_2$.query-size$()) \le n \wedge n \le value(t, group_1$.query-size$()) \wedge$
    Call$(t+1, group_1$.modify-group$(group_2, -n)) \wedge$
    Call$(t+1, group_2$.modify-group$(group_1, n))]$

Allowing Boats to Move

The second step consists of forcing the boat to move to another randomly selected bank whenever anyone is onboard. The following method is added to BOAT:

Constraint move-boat(): If anybody is onboard a boat, the boat automatically moves to another (unspecified) BANK. The destination bank is unspecified, and will be constrained indirectly by the goal.

$$
\begin{aligned}
&\textbf{dep}\ \textbf{DisableInherited}(\text{CONTROLLER}, \mathsf{move\text{-}boat})\\
&\textbf{dep}\ [t]\ \neg\mathsf{override}(\mathit{controller}.\mathsf{class}, \mathsf{move\text{-}boat}, \text{CONTROLLER}) \wedge\\
&\quad \textbf{people\_at}(t, \text{GROUP}, \mathit{value}(t, \mathit{boat}.\mathsf{query\text{-}onboard}())) > 0 \rightarrow\\
&\quad \exists \mathit{bank}[[t]\ \mathit{boat}.\mathsf{query\text{-}pos}() \not\hat{=} \mathit{bank} \wedge\\
&\qquad\quad \textbf{Call}(t, \mathit{boat}.\mathsf{move\text{-}to}(\mathit{bank}))]
\end{aligned}
$$

Additional Control: Don't be Stupid

In addition to the nondeterministic choice of actions provided by the methods above, it is also possible to introduce some more "intelligence" in the controller by adding further constraints on the acceptable state sequence.

There is no point in allowing a state to repeat.

Constraint no-repetitions(): At each timepoint, at least one group should change sizes.

$$
\begin{aligned}
&\textbf{dep}\ \textbf{DisableInherited}(\text{CONTROLLER}, \mathsf{no\text{-}repetitions})\\
&\textbf{acc}\ [t]\ \neg\mathsf{override}(\mathit{controller}.\mathsf{class}, \mathsf{no\text{-}repetitions}, \text{CONTROLLER}) \rightarrow\\
&\quad \exists \mathit{group}.\mathit{value}(t, \mathit{group}.\mathsf{query\text{-}size}()) \neq \mathit{value}(t+1, \mathit{group}.\mathsf{query\text{-}size}())
\end{aligned}
$$

There should be at least one person on the boat, except at the first and last timepoint in the plan. This avoids plans where everyone leaves the boat but nobody else boards it, leaving it empty for a period of time.

Constraint boat-not-empty(): There should be someone on the boat.

$$
\begin{aligned}
&\textbf{dep}\ \textbf{DisableInherited}(\text{CONTROLLER}, \mathsf{boat\text{-}not\text{-}empty})\\
&\textbf{acc}\ [t]\ \neg\mathsf{override}(\mathit{controller}.\mathsf{class}, \mathsf{boat\text{-}not\text{-}empty}, \text{CONTROLLER}) \rightarrow\\
&\quad \forall t. t > 0 \wedge t < t_* - 1 \rightarrow \sum_{\substack{\{g \mid g \in \text{GROUP} \wedge \\ [t]\, g.\mathsf{query\text{-}pos}() \hat{=} \textbf{onvera}\}}} \mathit{value}(t, g.\mathsf{query\text{-}size}()) > 0
\end{aligned}
$$

4.9.2 Additions for the Elaborations

Although the controller presented above is sufficient for the original version of the Missionaries and Cannibals domain, some of the elaborations alter basic properties of the domain and require further elaborations of the controller.

One Oar on Each Bank (#5)

In the fifth elaboration, there is one oar on each bank. To solve this problem, a cannibal must row alone to the other bank, pick up the second oar, and then row back. This means that there must be an interval of time where no groups change sizes, so no-repetitions must be modified in a new controller class OARCONTROLLER: If there is an oar in a position near the rowboat, then no groups have to change. An instance of OARCONTROLLER should then be created instead of an instance of CONTROLLER.

```
class OARCONTROLLER extends CONTROLLER
  obj ctrl : OARCONTROLLER
```

Constraint no-repetitions(): At each timepoint, at least one group should change sizes.

```
dep DisableInherited(OARCONTROLLER, no-repetitions)
acc $[t]\ \neg\mathsf{override}(\mathit{oarcontroller}.\mathsf{class}, \mathsf{no\text{-}repetitions}, \mathrm{OARCONTROLLER}) \rightarrow$
      $\exists \mathit{oar}.[t+1]\mathit{oar}.\mathsf{query\text{-}pos}().\mathsf{query\text{-}connection}(\mathit{rowboat}.\mathsf{query\text{-}onboard}()) \vee$
      $\exists \mathit{group}.\mathit{value}(t, \mathit{group}.\mathsf{query\text{-}size}()) \neq \mathit{value}(t+1, \mathit{group}.\mathsf{query\text{-}size}())$
```

In addition to this relaxation of no-repetitions, it is also necessary to extend the controller to take an oar whenever one is available.

Constraint take-oars(): If a rowboat is at a river bank where an oar is available, then the oar should be moved into the boat.

```
dep DisableInherited(OARCONTROLLER, take-oars)
dep $[t]\ \neg\mathsf{override}(\mathit{oarcontroller}.\mathsf{class}, \mathsf{take\text{-}oars}, \mathrm{OARCONTROLLER}) \wedge$
        $\mathit{oar}.\mathsf{query\text{-}pos}() \mathrel{\hat{=}} \mathit{rowboat}.\mathsf{query\text{-}pos}() \rightarrow$
      $\mathbf{Call}(t+1, \mathit{oar}.\mathsf{set\text{-}pos}(\mathit{rowboat}.\mathsf{query\text{-}onboard}()))$
```

The Bridge (#13)

If there is a bridge, the boat does not necessarily have to be used at all timepoints. The boat-not-empty constraint has to be disabled, which is done by overriding it in a new controller subclass BRIDGECONTROLLER without providing a new implementation.

```
class BRIDGECONTROLLER extends CONTROLLER
  obj ctrl : BRIDGECONTROLLER
 dep DisableInherited(BRIDGECONTROLLER, boat-not-empty)
```

The Boat Leaks (#14)

If the boat can leak, the controller must be extended to call the bail action at all timepoints.

```
class BAILCONTROLLER extends CONTROLLER
  obj ctrl : BAILCONTROLLER

  dep DisableInherited(BAILCONTROLLER, do-bail)
  dep [t] ¬override(bailcontroller.class, do-bail, BAILCONTROLLER) →
      Call(t, bailboat.bail())
```

The Boat Can Be Damaged (#15)

In elaboration 15, the boat can be damaged, and the action of moving to another river bank had to be split into two events: Moving to the river, and then after *crosstime* timepoints, arriving at the destination. The original controller states that groups must always change sizes from t to $t+1$, which clearly cannot be the case in this scenario. Instead, the groups must change sizes from time t to time $t + crosstime$, unless there was an emergency.

```
class SLOWCONTROLLER extends CONTROLLER
  obj ctrl : SLOWCONTROLLER

  dep DisableInherited(SLOWCONTROLLER, no-repetitions)
  acc [t] ¬override(slowcontroller.class, no-repetitions, SLOWCONTROLLER) ∧
      [t + 1, t + crosstime − 1]¬slowboat.query-emergency() →
      ∃group.value(t, group.query-size()) ≠ value(t + 1, group.query-size())
```

An additional precondition is required for move-boat: The controller should not call move-to for a boat when that boat is on the river.

```
  dep DisableInherited(SLOWBOAT, move-boat)
  dep [t] ¬override(slowcontroller.class, move-boat, SLOWCONTROLLER) ∧
        boat.query-pos() ≠ onriver ∧
      people_at(t, GROUP, value(t, boat.query-onboard())) > 0 →
      ∃bank[[t] boat.query-pos() ≠ bank ∧
           Call(t, boat.move-to(bank))]
```

4.9.3 Results

The timings in Table 4.1 on the next page were generated by the research tool VITAL (Kvarnström, 2005) using Java 1.3.1 and the HotSpot Server virtual machine on an 1800 MHz Pentium 4 machine. The total number of time steps in each plan is shown (including one step for initialization) together with the total amount of time required for generating the plan. Times are specified in seconds. We also provide some comparisons with the 10 elaborations implemented by Lifschitz (2000) in the Causal Calculator (McCain & the Texas Action Group, 1997), which was run on an unspecified machine.

Elaboration	Steps	Time (VITAL)	Time (CC)
Original	12	1.5	17.6
1	12	1.5	–
2	12	6.5	–
3	Unsolvable		
4	12	2.8	18
5	14	2.5	44
6	14	5.2	273
7	Unsolvable		
8	16	11.3	9746
9	12	7.8	22
10	6	1.7	–
11	12	2.3	55
12	12	1.8	–
13	5	1.6	2
14	12	1.7	9
15	36	5.2	–
16	16	165.5	1894
17	10	3.8	7361
18	14	24.0	–
19	12	16.6	–

Table 4.1: Test Results for the Missionaries and Cannibals Problems

The timings are *not* directly comparable and should not be taken as claims regarding the efficiency of the two approaches. This is especially true because (at least in VITAL) timings depend very much on the exact formulation of an elaboration, and could change drastically simply by altering the order in which objects are declared.

Two of the problems were unsolvable. We have not proved this within the logic: The logic-based controller used to solve the remaining 17 problems is not a full planner, and like the Causal Calculator, it requires as input the length of the plan to be generated. Proving that no plan (of arbitrary length) would solve these two problem instances would require additional reasoning outside the logic, for example by using depth first search with cycle checking. This procedure is complete due to the finite state space generated by any given problem instance and due to the fact that the applicability of an action only depends on the state in which it is invoked.

4.10 Traffic World

The object-oriented framework presented in this article has also been used for modeling the Traffic World scenario proposed in the Logic Modeling Workshop (Sandewall, 1999), previously modeled by Henschel and Thielscher (1999) using the Fluent Calculus (Thielscher, 1998). This domain consists of cars moving in a road network represented as a graph structure, together with a TAL-C controller class that "drives" a car.

4.11 Related Work

Much work has been done in combining ideas found in object-oriented languages with the area of knowledge representation. One such area is description logics (Borgida, Brachman, McGuinness, & Resnick, 1989; Brachman, Fikes, & Levesque, 1983), languages tailored for expressing knowledge about concepts (similar to classes) and concept hierarchies. They are usually given a Tarski style declarative semantics, which allows them to be seen as sub-languages of predicate logic. Starting with primitive concepts and roles, one can use the language constructs (such as intersection, union and role quantification) to define new concepts and roles. The main reasoning tasks are classification and subsumption checking.

Description logic hierarchies are very dynamic, and it is possible to add new concepts or objects at runtime that are automatically sorted into the correct place in the concept hierarchy. Some work has been done in combining description logics and reasoning about action and change (Artale & Franconi, 1998).

The modeling methodology presented in this article uses a different kind of class hierarchy that is fixed at translation time. Classes are explicitly positioned in the hierarchy, and classes and objects cannot be constructed once the narrative has been translated. Also, description logics do not use methods or explicit time, both of which are essential in the work presented here.

The approach presented in this chapter bears more resemblance to object-oriented programming languages such as Prolog++ (Moss, 1994), C++ or Java. In most such languages, however, a method is a sequence of code that is procedurally executed when the method is invoked. In our approach, a method is a set of rules that must be satisfied whenever the method is invoked. Since delays can be modeled in TAL-C, methods can be invoked over intervals of time and complex processes can be modeled using methods. It is also possible to invoke multiple methods concurrently.

An interesting approach to combining logic and object-orientation is Amir's object-oriented first-order logic (Amir, 1999, 2000), which allows a theory to be constructed as a graph of smaller theories. Each subtheory communicates with the other via interface vocabularies. The algorithms for the object-oriented first-order logic suggest that the added structure of object-orientation can be used to significantly increase the speed of theorem proving.

The work by Morgenstern (1998) illustrates how inheritance hierarchies can be used to work with industrial sized applications. Well-formed formulas are attached to nodes in an inheritance hierarchy and the system is applied to business rules in the medical insurance domain. A special mechanism is used to construct the maximally consistent subset of formulas for each node.

4.12 Conclusions

This article has presented a way to do object-oriented modeling in an existing logic of action and change, allowing large domains to be modeled in a more systematic way and providing increased reusability and elaboration tolerance.

The main difference between our work and other approaches to combining knowledge representation and object-orientation is due to the explicit timeline in TAL. Methods can be called over time periods or instantaneously, concurrently or with overlapping time intervals. Methods can relate to one state only or describe processes that take many timepoints to complete.

Although a few new macros have been introduced in this article, those macros are merely syntactic sugar serving to simplify the construction of domain descriptions. Thus, the most important contribution is not the syntax but the structure that is enforced on standard TAL-C narratives to improve modularity and reusability. It is also reasonable to believe that the added structure could be used to make theorem proving in $\mathcal{L}(\mathrm{FL})$ more efficient, although the current version of VITAL does not take advantage of this.

4.13 Acknowledgements

This research is supported in part by the Swedish Research Council for Engineering Sciences (TFR), the WITAS Project under the Wallenberg Foundation and the ECSEL/ENSYM graduate studies program.

Part III

TALplanner

Chapter 5

Planning

Up to this point, the work presented in this thesis has generally assumed the existence of a predefined set of actions to be performed. The new methodology for modeling qualifications in TAL requires all action occurrences to be specified in advance. Our work on using object-oriented modeling techniques to increase elaboration tolerance does use a simple "controller", based on fluent dependency constraints with incompletely specified trigger conditions, which can be said to autonomously determine which actions to perform – but this technique was only intended as a proof of concept, and cannot be expected to be as efficient or as versatile as a true *planner* built for the express purpose of determining which actions should be performed in order to achieve a predetermined goal.

This chapter begins by briefly describing the planning problem[1], the concepts involved in formally modeling a planning domain, and the different levels of expressivity that may be required to model planning domains of varying complexity. We continue by describing forward-chaining planning, one of several common approaches to solving the problem of actually finding a plan. The last section of this chapter describes TLPlan (Bacchus & Kabanza, 2000), a forward-chaining planner based on the idea of allowing the user to specify additional control information in the shape of temporal formulas that help guide the planner towards the goal.

The introduction to planning in this chapter is intended to provide the necessary background for the presentation of TALplanner, a new planner where planning domains are modeled in TAL and where the search strategy is based on the ideas pioneered by TLPlan. TALplanner itself is the topic of the remaining chapters. As indicated below, these chapters are loosely based on a number of published papers and articles on TALplanner, though much of the material is extensively rewritten and some is completely new.

[1]See also Ghallab, Nau, and Traverso (2004) for a complete presentation of the state of the art in planning.

Chapter 6 shows how concepts related to planning domains and planning problem instances can be modeled in TAL. Two different methods for representing control formulas in TAL are presented. The first method involves using purely TAL-based control formulas together with a new formula evaluation framework developed to allow such formulas to be tested efficiently and incrementally in the state sequence generated by a plan candidate. The second method is based on introducing tense macros into TAL, thereby enabling the use of a modified version of the progression algorithm used in TLPlan and facilitating a benchmark comparison between TALplanner and TLPlan. Some preliminary benchmark results from an early TALplanner article are provided (Doherty & Kvarnström, 1999; Kvarnström & Doherty, 2000b; Doherty & Kvarnström, 2001; Kvarnström, 2002).

The first version of TALplanner was restricted to generating sequential plans. Relaxing this restriction to generate concurrent plans involves changes to the search space traversed by the forward-chaining search algorithm. Perhaps more importantly, concurrency invalidates the assumption that each action in a plan is the only possible cause for change within its own execution interval. The interactions between concurrent actions must therefore be governed by stronger rules than those between sequential actions, in order to ensure that conflicting actions are not allowed to execute concurrently while still permitting an extensive use of concurrency where this does not have detrimental effects. Several different methods for controlling concurrency are discussed in Chapter 7, together with an extension to the modeling language to allow a succinct representation of resource constraints (Kvarnström et al., 2000; Kvarnström & Doherty, 2000b).

Chapter 8 demonstrates how significant performance improvements can be achieved by applying existing and new domain analysis techniques to control formulas. In some cases, these techniques also allow the planner to automatically transform part of a control formula into a precondition, permitting such conditions to be tested at an earlier stage in the planning process and thereby further decreasing the number of search nodes to be expanded (Kvarnström, 2002).

Chapter 9 contains the results from two international planning competitions, including benchmark comparisons with other state-of-the-art planners and descriptions of the control formulas that were developed for several of the competition domains in the most recent competition (Kvarnström & Doherty, 2000b; Kvarnström & Magnusson, 2003).

Chapter 10 concludes the TALplanner part of the thesis with a discussion of some of the research that has been done during the project and the lessons that have been learned.

5.1 Introduction to Planning

In its most general form, the planning problem can be defined as the task of determining what you need to do in order to achieve a given goal. Some possible goals could be "the drawer should be open", "block **A** should be on top of block **B**", "this 15-puzzle should be solved", "I should have an ice-cream", "all orders should be packed and mailed by tomorrow", "everyone who is currently in the burning building should be at least one block away", "there should be a human settlement on Mars", or "there should be world peace by noon tomorrow". The resulting *plan* should describe in some formal manner what to do in order to achieve the goal, most likely in terms of a set of actions that should be performed and some constraints on the order in which they should be performed.

Trying to find an algorithm that solves all instances of this rather general problem – an algorithm that always succeeds in finding a valid plan given any possible goal – might be slightly too ambitious, though. As a first step it is therefore necessary to define and formalize a more constrained version of the planning problem.

5.1.1 Expressivity versus Domain Dependence

If the planning problem as expressed above is too general, the question is to what extent it should be constrained. This discussion will be facilitated by the introduction of two concepts: Planning domains and problem instances. These concepts will be discussed in more detail later in this chapter.

A *planning domain* is characterized by a set of general concepts relevant for solving an entire class of related problem instances. For example, the previously mentioned goal "block **A** should be on top of block **B**" may be taken from the standard blocks world domain, which is characterized by the existence of a table and a set of blocks, by the fact that any block may be either on the table or on top of another block (allowing towers of blocks to be formed), and by the ability to pick up blocks and move them to the table or place them on top of other blocks. Note that no mention is made of which blocks are present or which blocks are on top of each other, only that there *are* blocks and that blocks *can* be on top of each other.

A *problem instance* is a concrete problem within a particular planning domain. The specification of a problem instance includes the entities or objects involved, a set of initial conditions, and a goal that should be achieved. In the blocks world domain, for example, a problem instance specification may state that there are four blocks named **A**, **B**, **C** and **D**, that all blocks are initially on the table, and that the goal is to place the blocks in two towers where **A** is on top of **B** and where **C** is on top of **D**.

Clearly, an algorithm which is only capable of generating plans for one particular problem instance is not interesting and should not be considered to be a planner. An algorithm limited to a single fixed planning domain might still be both interesting and useful, though, as long as it is applicable to all problem instances for this domain. For example, there are planners specific to the blocks world domain, which are able to generate action sequences that transform a certain configuration of blocks into another more desirable configuration but are incapable of handling even minor changes to the domain without modifying the planner itself (for example Kibler & Morris, 1981; Gupta & Nau, 1992; Slaney & Thiébaux, 2001). Such *domain-dependent* or *domain-specific* planners can use algorithms and data structures strictly adapted to a single domain, which can enable numerous optimizations for speed as well as memory usage. On the other hand, much of the work that goes into optimizing a domain-dependent planner can be rendered obsolete if any of the basic assumptions made about the domain were to change.

Most research in the field of planning is not concerned with domain-dependent planners, but with general planners where a formal description of the planning domain is part of the input to the planning algorithm. Numerous formalisms for describing planning domains as well as problem instances have been developed over the years, some of the more well-known ones being STRIPS (Fikes & Nilsson, 1971), ADL (Pednault, 1989), the Action Description Language, and a number of variations of PDDL, the Planning Domain Definition Language (Ghallab, Howe, Knoblock, McDermott, Ram, Veloso, Weld, & Wilkins, 1998; Fox & Long, 2003; Edelkamp & Hoffmann, 2004).

Planners with the ability to take a domain description as input are generally called *domain-independent planners*. It is important to realize that this distinction between domain dependence and independence is not as clear-cut as it may seem at first glance. There is considerable variation in the expressivity permitted in the domain definition languages of different planners. More expressive languages allow a wider range of domains to be modeled, and could therefore be said to entail a higher degree of domain-independence.

The differences in expressivity between various domain definition languages become even more apparent if we take the view that expressivity is measured in terms of what can be modeled *conveniently* in a given language, rather than in terms of what planning problems can be solved through potentially complex transformations of planning problems into this language (though the concept of "convenient modeling" would admittedly be difficult to formalize). For example, an operator with conditional effects can be emulated using multiple operators where each precondition corresponds to the original precondition together with a combination of effect conditions, but this requires a transformation that generates an exponential number of operator types. Similarly, some planners allow the use of a finite subset of the integers and provide direct support for common numeric operations and relations. Even in planners that do not directly support integers, numbers *can* be modeled using explicitly enumerated constants (**n0**, **n1**, **n2**, ...) together with explicitly enumerated definitions of any relations and functions that may be required for those constants (lessthan(**n0**,**n1**), plus(**n2**,**n2**,**n4**), and so on), but in practice this is only feasible for smaller subsets of the integers.

5.1.2 Classical Planning

Even though few planners share exactly the same level of expressivity, there are some constraints that could be said to be common for the majority of planners in the literature. Most of these constraints were influenced by the early STRIPS planner (the Stanford Research Institute Problem Solver, Fikes & Nilsson, 1971), and these constraints define what is often called *classical planning*. Though not even this concept is completely well-defined, there are some common properties that are almost universally agreed upon.

Classical planning uses a finite state-based representation of the world, where the agent has complete and correct knowledge about the initial state of the world and the preconditions and effects of the actions it can perform. Actions are viewed as single-step operations, disregarding any temporal structure within the execution of an action. Goals in classical planners can only constrain the final state that results from executing a plan. Constraints on what happens before that state – for example, that before one ends up at the goal location one should also visit two other locations in a certain order – generally cannot be modeled in a natural manner.

Another assumption in classical planning, which is so basic that it is sometimes not even stated explicitly, is that nothing changes in the world except when the agent executes an action. This means that there is no need for the more complex plan structures used in conditional plans, where sensor actions can be used in conjunction with conditional statements and loops in order to make a plan adaptable to currently unknown aspects of the eventual execution environment: The agent has complete information about the initial state and can determine exactly what would happen if a certain potential plan would be executed, without having to consider the actions of other agents or events that occur naturally in the world. Given this assumption, planning can be separated from execution, and there is no need to consider execution monitoring (verifying that the intended effects actually materialize and that no required conditions are violated), replanning (creating a new plan in case an unexpected event causes the original plan to fail), or plan repair (modifying the original plan appropriately in case of unexpected events, without starting over from the beginning).

In order to give an intuitive picture of what can be done using classical planning, we will now present a small set of classical benchmark domains. We will then introduce some of the concepts and terminology involved in the field of planning using these domains as a source of concrete examples. At the same time, we will mention some of the basic limitations of classical planners as well as a number of possible extensions and relaxations that can be made in order to support the modeling of more complex domains and more elaborate plan structures. Many of these extensions have already been implemented in various planning algorithms in the literature. Note, however, that the intention is not to give a definition of the boundary between classical and non-classical planning – again, there is no consensus on the precise limits of classical planning, and no such definition is required for the

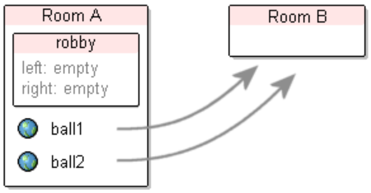


Figure 5.1: A Gripper Problem Instance

work presented here. Neither is the spectrum of possibilities discussed below intended to be exhaustive. For example, concepts only used by hierarchical planners will not be covered.

5.1.3 Planning Domain Examples

The following three planning domains are commonly used as benchmark domains in the literature. Since we have not yet introduced a formalism for describing planning domains, the descriptions below will be somewhat informal, but should still serve to give an intuitive understanding of the domains. Additional planning domains will be presented in Chapter 9.

Example 5.1.1 (Gripper Domain)
The *gripper domain* is one of the simplest planning domains in the literature. As such, it has been used widely in benchmark testing and exists in several variations with numerous minor differences. In what is probably the most common variation there is a single robot with one or more grippers, which can be used to pick up or drop objects (typically balls). There are two rooms, and the robot can move freely between the rooms. In the initial state, all objects are located in one of the rooms, usually called room A. The goal requires all objects to be in the other room, usually called room B.

Figure 5.1 shows a problem instance from a variation of the gripper domain where the robot, **robby**, has two grippers, **left** and **right**. The problem instance is quite tiny, containing only two rooms and two balls, both of which are initially in **roomA** and both of which should eventually end up in **roomB**. ■

Example 5.1.2 (Blocks World)
The *blocks world*, originally due to Winograd (1972), consists of a finite number of blocks together with a table and a single crane which can be used to move the blocks. Each block is either on the table or on top of another block. There is enough space for all blocks to be on the table simultaneously, though there cannot be more than one block on top of a given block at any given time. Blocks can be stacked in towers of arbitrary height.

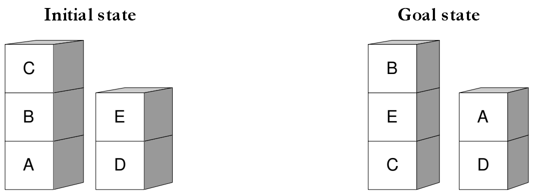


Figure 5.2: A Blocks World Problem Instance

The initial state of a blocks world problem instance provides a unique configuration of blocks. The goal may be to move the blocks into another unique configuration, or it may specify the location of some blocks (**A** must be on top of **B**) while ignoring the location of other blocks (**B** can be on the table or on another block).

An example of completely specified initial and goal configurations for a small five-block problem instance can be seen in Figure 5.2. ■

Example 5.1.3 (Logistics Domain)
The standard *logistics domain* contains a number of cities, each of which contains one or more locations. At some of these locations, there may be packages. The packages can be transported between locations in the same city using trucks. Some locations are airports, and there are a number of airplanes that can be used to transport packages between different airports.

The goal is usually to deliver each package from its initial location to its destination. In the worst case, each object may have to be transported by truck from its original location to an airport, by airplane to another airport, and then by truck from that airport to its final location, thus requiring up to nine actions per package when loading and unloading actions are included.

Figure 5.3 contains a small example problem instance with four cities (**city-1** through **city-4**), eight locations (two in every city), six packages to be transported (**package-1** through **package-6**), four trucks (**truck-1** through **truck-4**), and one airplane (**plane-1**). Though not indicated in the figure, four of the locations are airports, one in each city: **city1-2**, **city2-2**, **city3-2**, and **city4-2**. The arrows in the figure indicate intended destinations for each package; note that **package-1** is intended to remain in its initial location **city2-1**. In this example, there are no destinations specified for trucks or airplanes. ■

It is easy to dismiss these domains as being too simplified compared to real-world problems. The logistics domain, for example, does not model space restrictions in trucks and airplanes, timetables for airplanes, fuel costs, driver availability, or even the amount of time required to move between two locations. Nevertheless, the domains are both important and useful, even for planners expressive enough to handle more elaborate versions of the same problems, because they provide concrete examples of certain structures that also occur in more complex domains.

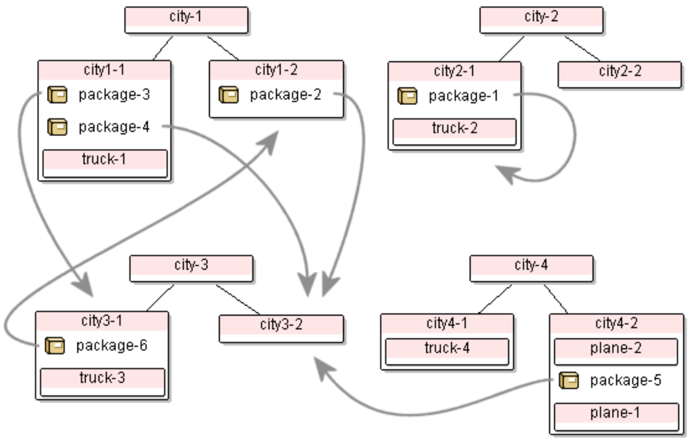


Figure 5.3: A Logistics Problem Instance

The gripper domain provides the quintessential example of a planning domain with a high degree of redundancy. For example, as long as the goal is for *all* balls to end up in the second room, it does not matter which ball the robot picks up first, and it does not matter which gripper is used. This means that the operator pick$(ball, gripper)$, with one instance for each combination of ball and gripper in a problem instance, could in essence be condensed into the operator pick-random-ball-in-random-gripper, which only has one instance regardless of the size of the problem instance, as long as the planner is clever enough to detect that balls and grippers are interchangeable.

In the blocks world, even when the planner has managed to satisfy *almost* all ground facts required by the goal, it may still be very far from actually reaching a goal state – because even if a tower of blocks is “almost” correct, a single discrepancy at the bottom will force you to tear down the entire tower in order to make it possible to repair the problem. This is especially difficult for some planners because in some sense it requires moving *away* from the intended goal, moving blocks that were already on top of their intended destination blocks.

Finally, the logistics domain provides an example of a highly concurrent domain, where failing to make use of this concurrency (such as driving more than one truck at the same time) will lead to a very inefficient plan.

5.1.4 Describing the World: States, State Variables, and Objects

In classical planning, it is assumed that the dynamic world for which we are creating plans can be described in terms of a finite set of *state variables* which can generally take a number of arguments. In the blocks world, for example, there is usually a boolean state variable called on$(block_1, block_2)$ which takes on the value **true** if

$block_1$ is on top of $block_2$ and the value **false** otherwise. In the logistics domain, $\mathsf{in}(obj, truck)$ might be used to represent the fact that a certain package is inside a certain truck.

Some planners are limited to such *boolean* state variables, which could be viewed as *predicates* in first-order logic. This means that the color of a block has to be modeled as a predicate $\mathsf{has\text{-}color}(block, col)$, and one has to take care that a block is not assigned two colors at once. Other planners also allow you to directly model non-boolean properties, where $\mathsf{color\text{-}of}(block)$ might be a function whose value is the color of the given block.

If state variables can take arguments, there is usually also a well-defined finite set of objects that can be used to instantiate the arguments. In a logistics planning problem, for example, there might be exactly four trucks (**truck-1** through **truck-4**) and six packages (**package-1** through **package-6**). For some planners, there is only one object domain, and state variables are untyped. This would mean that both in(**package-1**, **truck-1**) and in(**truck-2**, **truck-1**) would be possible, though being able to model the latter fact might not be very useful. Most modern planners do allow the use of types, permitting the user to specify that in takes a `package` and a `truck` as arguments.

A *state* is essentially an instantaneous snapshot of the world, and provides a value for each of the state variables. Sometimes this world is an artificial one, as when we are creating plans to be executed in a simulation inside a computer, and then the state variables may provide a complete description of all aspects of the world. Other times, the dynamic world for which a plan should be generated may in fact be the real world, in which case a state can obviously only be an abstract model of the true state of the world. The expressive power of the planning formalism being used provides an upper bound for the level of detail that can be achieved in the model, though it is common to use a less precise model than the formalism allows, either because additional detail is not necessary for the task at hand or because additional detail would be prohibitively expensive in terms of time and space requirements for the planner. For example, roads and road networks are rarely modeled in the logistics domain, even though this would not pose a problem in common domain definition languages such as STRIPS, ADL and PDDL. Instead, trucks are usually modeled as being able to move between arbitrary locations within a city in a single time step, most likely because logistics problems have historically been difficult to solve with fully automated domain-independent planners even when this artificial abstraction is applied.

Some planners make explicit use of states, for example using methods such as forward-chaining to find solution plans by searching the state space corresponding to a planning problem instance (Section 5.2). This is not a requirement, though: There is a wide variety of other planning paradigms, some of which never work with complete states at all, although an overview of these paradigms is outside the scope of this thesis. Nevertheless, the concept of "state" is often useful in understanding the work performed by a planner.

Defined Predicates

In addition to ordinary state variables, which are given a value in the initial state and are updated using action effects, some planners also support the use of variables that are defined in terms of other variables. Such state variables are sometimes called *defined predicates*, as opposed to the ordinary *basic* or *primary* predicates.

As an example, consider a version of the gripper domain where there are multiple robots, each of which has multiple grippers. This may be modeled in terms of basic predicates such as has(*robot*, *gripper*) and is-carried-in(*object*, *gripper*). The secondary predicate is-carried-by(*object*, *robot*) can then be defined to hold exactly when there exists a gripper which belongs to the robot and which is holding the object, without the need for action effects that explicitly update the is-carried-by predicate each time the has or is-carried-in predicates are updated. Creating such secondary concepts may make it easier to create a domain definition as well as easier to understand it once it has been written, especially for more complex domains where an entire hierarchy of secondary concepts may be used.

5.1.5 The Beginning of Time: The Initial State

Many planners require complete information about the initial state of the world – the state of the world just before the plan that is being generated would be executed.

This information is often provided to the planner in the shape of a set of positive ground literals that hold in the world. The *closed world assumption* is then applied: All literals that are not explicitly mentioned to be true are assumed to be false. This is mainly a representational convenience, useful because most predicates usually have far more negative than positive instances. In the blocks world, for example, it is sufficient to state on(**A**,**B**) – block **A** is on top of block **B** – without having to explicitly list all the blocks that **A** is not on top of.

Some planners instead use a more general first-order representation allowing (and requiring) the specification of both positive and negative instances of literals.

There are also planners that allow incomplete information about the initial state. This is clearly more useful for real-world domains, where not all information may be known to the planner in advance, but leads to additional problems both in terms of representation (how to represent the knowledge that does exist) and in terms of added complexity during the search for a plan, as discussed briefly in Section 5.1.8.

5.1.6 What to Achieve: Goals

In classical planning, a goal is defined in terms of a set of acceptable goal states (or as a set of logical formulas characterizing the goal states). Any combination of actions that leads from the initial state to a goal state is a valid plan, regardless of what may happen along the way to that state. For the blocks world, the goal is a specific configuration of blocks, or possibly a set of acceptable configurations. In the logistics domain, the goal usually specifies destinations for all packages, while the final locations of airplanes and trucks are often left unconstrained.

One can also imagine other goal types than purely state-based ones. For example, some planners permit the specification of temporally extended goals that relate to the entire sequence of states generated by a plan. Such goals may include maintenance goals as well as safety goals, such as the goal that a robot should never visit a certain set of dangerous locations at any time during the execution of a plan. Other types of non-classical goals might include goals with temporal deadlines or goals which must be achieved at some point in the execution of a plan but which need not necessarily still hold in the final state.

5.1.7 Doing Something: Operators and Actions

In order to be able to create a plan with some level of confidence that the plan actually achieves a certain goal, an agent must have a reasonably good idea about what it can do – what actions it can perform, how those actions affect the external world, and how the effects vary given different initial conditions. In the most general case, the agent might be expected to infer this information by performing random actions and observing their effects, or it may at least be expected to handle occasional exceptions from general action descriptions when an action fails to have its intended effects. In classical planning, however, this is considered far too difficult. Instead, the agent is assumed to have a complete and correct description of all available actions.

There is no general agreement on the difference between operators and actions. The following terminology will be used here: An *operator* is a template that can take arguments and can be instantiated into a set of concrete *actions* by replacing argument variables with objects of the proper type. An action can also be called an *operator instance* for further emphasis on the fact that it is an instantiation of a template.

The blocks world commonly uses the operators $\mathsf{pickup}(\mathit{block})$, $\mathsf{putdown}(\mathit{block})$, $\mathsf{stack}(\mathit{block}_1, \mathit{block}_2)$ and $\mathsf{unstack}(\mathit{block}_1, \mathit{block}_2)$, although there are also versions using a smaller number of more powerful operators. The first operator has instances such as $\mathsf{pickup}(\mathbf{A})$ and $\mathsf{pickup}(\mathbf{B})$, if **A** and **B** are objects of type `block` in the current problem instance. The logistics domain often uses the operators $\mathsf{load\text{-}truck}(\mathit{truck}, \mathit{obj})$ and $\mathsf{load\text{-}plane}(\mathit{plane}, \mathit{obj})$ to load packages into vehicles, $\mathsf{unload\text{-}truck}(\mathit{truck}, \mathit{obj})$ and $\mathsf{unload\text{-}plane}(\mathit{plane}, \mathit{obj})$ to unload packages, and $\mathsf{drive\text{-}truck}(\mathit{truck}, \mathit{loc}_1, \mathit{loc}_2)$ and $\mathsf{fly\text{-}airplane}(\mathit{plane}, \mathit{airport}_1, \mathit{airport}_2)$ to drive or fly vehicles between two locations.

Each operator has a *precondition* that determines which of its instances the planner is allowed to invoke in any given state. For example, a package can only be loaded into a truck if the package and the truck are at the same location, and you can only pick up a block if it is at the top of a tower and you are not already holding a block. In some planners this condition is limited to being a simple conjunction of literals, while other planners allow arbitrary first-order logic formulas, including disjunctions and quantifiers. The precondition may be constrained to referring to the state where the action is invoked, or it may be allowed to refer to the entire interval of time during which the action is being executed.

Each operator definition must also specify exactly how the world is affected when it is invoked, in terms of a set of *effects*.

In classical planning, all effects are assumed to be deterministic: Given complete knowledge about the state where the action is invoked, it is possible to predict exactly how the action will affect the environment. For example, invoking put-down(*block*) in the blocks world will definitely cause ontable(*block*) to become true and holding(*block*) to become false. Less restricted planners may also want to deal with non-deterministic or probabilistic effects.

Many planners allow the use of *conditional effects*, where some effects only take place if a certain condition holds in the state where an operator instance is invoked. These conditions should not be confused with preconditions. If the precondition of an action does not hold, the action must not be invoked. If the condition of a conditional effect does not hold, the action can be invoked, but this particular conditional effect will not take place.

Conditional effects permit operators to have different effects when invoked in different world states. In many domains, though, it may be difficult to determine in advance the exact effects an operator would have in *all* possible states. Assume, for example, a variation of the logistics domain where trucks have limited carrying capacities. What would happen if one attempted to load additional packages into a truck which according to our model has no remaining space? Perhaps the package would still fit into the truck, because our model was pessimistic and there was in fact some additional space available. If not, perhaps there is a robotic package loader which will try to follow the plan and load the package despite lacking sufficient space. What would happen then? Would the package fail to be loaded, or would it be pushed into the truck with enough force to damage the package itself or some of the previously loaded packages? Would some packages that were already loaded fall out of the truck?

Clearly, this relates to the ramification and qualification problems in the area of reasoning about action and change. In the planning area, these problems are sidestepped by only allowing operators to be invoked in situations where their preconditions are known to hold. By strengthening the preconditions of an operator to a sufficient degree, the question of what would happen under any other circumstances is rendered irrelevant. For example, if the preconditions guarantee that one only attempts to load packages into trucks that are known to have sufficient remaining carrying capacities, the planner does not need to know what would happen if it tried to load packages into trucks that are already full.

In classical planning, the *internal temporal structure* of an operator is quite limited: Operators are assumed to be without temporal constraints, and simply model a single state transition from an invocation state to an effect state. Some planners go beyond the classical framework and allow actions to have duration. This is becoming more and more common, perhaps partly because actions with duration were required in the Third International Planning Competition (IPC-2002, Long & Fox, 2003). The duration of an action may be fixed, or it may be determined by the arguments and the state in which it is invoked: The time required to fly between two cities depends on the speed of the airplane and the distance between the two cities. Some planners also support actions for which the planner can freely choose the duration, possibly within a given bound.

Some planners allow effects to take place at multiple points in time during the execution of the action, as opposed to only taking place at the start and end of the action. This richer temporal structure may in some cases be irrelevant when sequential plans are generated, because in that case only the final state resulting from invoking an operator is important. On the other hand, it can be absolutely necessary when modeling concurrent domains where interactions between concurrent actions have to be taken into account, because of possible conflicts or synergistic effects. Planners that do not directly support effects at arbitrary timepoints must instead model such effects using multiple actions or other mechanisms.

5.1.8 Combining Actions into Plans

In our terminology, a *plan* is an executable set or sequence of actions. A plan which also achieves the goal of a particular problem instance will be called a *solution* or *solution plan*.[2]

Since there can be dependencies between the actions that make up a plan, there is a need for some kind of temporal structure determining the order in which these actions should be performed. A large variety of temporal plan structures have been used by planners in the literature, ranging in complexity.

As noted above, many classical planners are limited to single-step actions where there is no duration associated with the execution of an action.

Some of these planners generate a totally ordered sequence of actions (Figure 5.4, top row). There are domains for which this is a reasonable restriction which does not have a significant impact on the quality of the resulting plans. For highly concurrent domains such as the logistics domain, though, actually executing such a sequential plan would be highly inefficient, with at most one vehicle being allowed to move at any given time.

Other classical planners generate a partially ordered set of single-step actions (Figure 5.4, second row), where the agent is free to execute certain actions in parallel or in arbitrary order as long as certain other actions have already been executed. Graphplan-style planners (Blum & Furst, 1997; Koehler, Nebel, Hoffmann, & Dimopoulos, 1997) generate plans consisting of a totally ordered sequence of action sets, where the actions within an action set can be executed in arbitrary order as long as they are executed before any action in the next set (Figure 5.4, third row).

[2]In some articles and papers in the literature, only solutions are called plans.

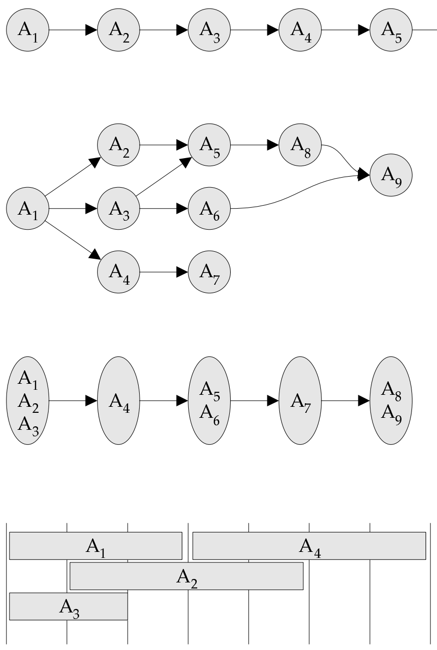


Figure 5.4: Plan Structures

Planners which can use actions with duration may need more complex plan structures, where each action is annotated with the temporal interval where it is to be executed (Figure 5.4, fourth row). Such structures could be extended further by allowing temporal intervals to be only partially specified, or by permitting specifications of minimum or maximum temporal distances between two actions in the plan.

Classical planners assume complete knowledge about the world, but in the real world, uncertainty is quite common. In order to improve the fidelity of the planner's approximation of reality, it may be necessary to add support for incompletely specified initial states or incompletely specified action effects. If the agent must deal with such *incomplete knowledge* the plan structure may also have to change, depending on the technique chosen to deal with this incompleteness. In *conformant planning*, the planner should find an ordinary sequential plan where it can be guaranteed that the goal will be achieved given the knowledge which is available to the planner before the execution phase begins, by making sure that no actions in the final solution plan depend on unknown information. In *conditional planning*, the agent also has access to sensing actions which can be used to increase the knowledge of the agent during the execution phase. Conditional plans may include branching, where different actions should be performed depending on the actual state of the world, which may in turn depend on the actual outcome of earlier actions performed by the agent. Conditional plans may also contain loops, where a conditional subplan may have to be executed multiple times until some condition is satisfied. In the remainder of the thesis we will not deal with sensing actions or incomplete knowledge.

Ideally the choice of plan structure should be defined according to the abilities of the agent or agents which will eventually execute the plan. In the logistics domain, for example, each vehicle is clearly able to move independently of other vehicles, and the potential for concurrency ought to be taken advantage of. In a more elaborate version of the standard logistics domain where distances are modeled, the time required to drive between two locations would be strongly dependent on the distance between the locations, requiring the use of partially overlapping actions as opposed to Graphplan-style concurrency in order to create reasonable plans.

In some cases using a plan structure able to express any plan the agent can execute might be computationally infeasible (at least for current planning algorithms), and then it might be necessary to use a different temporal structure which is less suited to the agent but which at least makes it possible to find a solution. However, it should also be noted that using a more elaborate plan structure could actually be a computational advantage in certain cases: If the set of possible candidate plans is strictly larger, there are more alternatives open to the planner, which could potentially make the task of finding a solution easier.

5.1.9 Plan Quality Criteria

For most non-trivial domains, there are a multitude of ways that any reachable goal can be achieved. Some subtasks are independent of each other and can be performed in any order. Some objects in the domain may be functionally identical, so that it does not matter which object is used – for example, if there are two empty trucks available at the same location in a logistics problem instance and only one is required, either truck could be used with identical results. Also, for many actions there is an inverse action with the opposite effects, and arbitrarily many instances of these actions can be added to a plan. Simple cycle checking removes some of those plans, but not those where other actions take place inbetween: When a truck in the logistics domain drives from **A** to **B**, a package is loaded into another truck, and the first truck drives back from **B** to **A**, this leads to three distinct states with no easily detectable cycle.

Consequently, it is usually possible to find a large number of solutions satisfying any given reachable goal, and usually some of them are "better" in some sense and should be preferred over others. In the most general case, we want to find a solution which *optimizes* a given function. For example, in an extended logistics domain where fuel is modeled, we might want to find a solution that minimizes fuel usage while still achieving all goals. The level of support for optimization varies greatly between different planners.

Quite a few planners unconditionally try to find a solution containing as few actions as possible, and do not allow the user to specify a function to be minimized or maximized. Some of these planners guarantee that the resulting plan will be optimal in the number of actions, while others only make a reasonable attempt to create a short plan.

Graphplan-style planners (Blum & Furst, 1997; Koehler et al., 1997), which usually do not support explicit action durations, guarantee that a generated plan contains a minimum number of temporal steps but ignore the number of concurrent actions within each temporal step. In other words, such planners prefer to return a plan with 150 actions where the inherent parallelism in the domain allows the plan to be executed in 10 steps, rather than a plan with 40 actions partitioned into 11 temporal steps.

Some planners which support explicitly specified action durations guarantee that a minimum-duration plan is found, even though this plan might contain a larger number of actions than other valid plans. The Graphplan approach has also been extended in this manner (Smith & Weld, 1999).

Still other planners permit the user to specify a function that should be minimized, where this function may refer to the length of the plan or to other properties such as fuel usage.

5.1.10 Domains and Problem Instances

We have now discussed a number of different types of information that must be available to a planner in order to solve a specific planning problem.

Some of this information relates to the general structure of the world for which plans should be created. This information is usually considered to define a *planning domain*. For classical planners, the domain specification defines a set of state variables, a set of operators (and their preconditions and effects), and for planners with typed objects, a set of object types. This is similar to the narrative background specification (NBS) in TAL (Section 2.3), with the exception that the NBS also defines the sets of objects belonging to each type.

The rest of this information is far more likely to change across different invocations of the planner. This information defines a *planning problem instance* for the given domain, and includes a specification of the initial state, the goal, and the actual (possibly typed) objects. This is similar to the narrative specification (NS) in TAL.

5.2 Forward-Chaining Planning

Given a planning domain description and a planning problem instance, it is not immediately obvious what would be the best way to find a plan that achieves the given goal. A large number of different methods have been investigated, but somewhat surprisingly, one of the most straight-forward approaches – forward-chaining – has had a renaissance during the last few years, with planners such as HSP (Bonet & Geffner, 1998), FF[3] (Hoffmann & Nebel, 2001), and TLPlan (Bacchus & Kabanza, 2000) providing better performance and higher expressivity than many earlier state-of-the-art planners that used more complex search methods.

The search space for a forward-chaining sequential planner can be viewed from at least two similar but somewhat different perspectives.

First, the search space can be defined as a graph where each node is a *state* and where there is an edge between two nodes (states) n_1 and n_2 iff there exists an action that would lead from n_1 to n_2. The graph induced by this definition is naturally directed, but it is not acyclic. In the blocks world, for example, it is possible to pick up a block and immediately put it down again, leading to a cycle of length 2 in the search graph.

Second, the search space can be defined as a tree where each node is a *plan* (Figure 5.5, described below). This tree can be defined inductively: The empty plan is necessarily executable and must therefore belong to the tree, and given any plan p in the tree, if the action a is applicable in the state resulting from executing p, then $\langle p; a\rangle$ is a child of p. In other words, the root node corresponds to the empty plan, and each subsequent node is generated by appending exactly one action to the plan of the parent node. Note that according to this definition, the same final state can be associated with many different nodes within the search tree, if that state can be reached through more than one possible sequence of actions. This definition will be used in the remainder of the thesis.

Figure 5.2 on page 119 shows an example problem instance from the blocks world, with an initial configuration of blocks on the left hand side and a goal configuration on the right hand side. The initial configuration is replicated in Figure 5.5, which also shows part of the search space for this problem instance; due to space considerations, we do not display complete plans in each node but instead display the new action being added in each node as a label on the incoming edge, with abbreviated operator names. In the initial node (the empty plan), the planner is visiting the initial state, where only two actions are applicable: $\mathsf{unstack}(\mathbf{C}, \mathbf{B})$ and $\mathsf{unstack}(\mathbf{E}, \mathbf{D})$. If the first action is applied, three new actions become applicable: $\mathsf{putdown}(\mathbf{C})$, $\mathsf{stack}(\mathbf{C}, \mathbf{B})$, and $\mathsf{stack}(\mathbf{C}, \mathbf{E})$. Note that the initial node and $\langle\mathsf{unstack}(\mathbf{C}, \mathbf{B}); \mathsf{stack}(\mathbf{C}, \mathbf{B})\rangle$ are different nodes, but both nodes generate the same final state, since the second action undoes the effect of the first action. The node label "C" indicates that a state cycle can be detected. Given the use of single-step actions and given that only the invocation state of an action is relevant to the applicability of that action, any plan containing a state cycle is redundant: There must exist a corresponding shorter plan where the action subsequence generating the state cycle has been removed. Thus, any plan containing a state cycle can be pruned from the search tree.

[3]http://www.mpi-sb.mpg.de/~hoffmann/ff.html

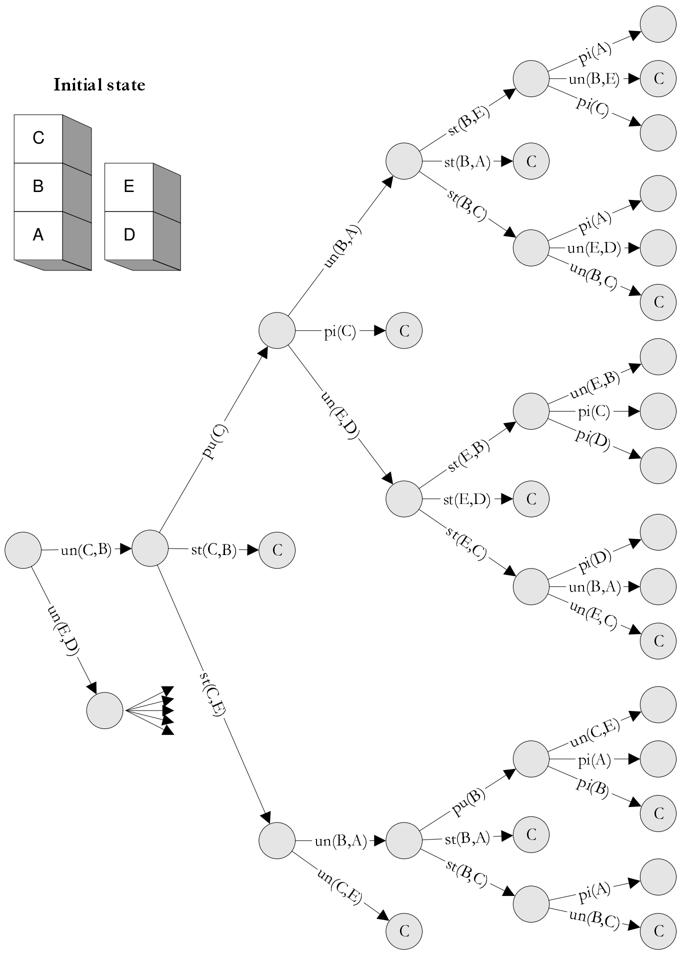


Figure 5.5: Part of the Forward Chaining Search Space for Figure 5.2

Forward-chaining planners begin at the initial node and search the tree for a path leading to a goal state. This very straight-forward paradigm has a definite advantage: At any point in the search process, the planner always has a complete description of the entire sequence of states generated by the current plan candidate – at least this is the case if the initial state is completely defined and plan operators are deterministic, which is always the case in classical planning. This, in turn, facilitates the use of complex operator types with quantified conditional effects, disjunctive preconditions, and indirect effects, which can be considerably more difficult to add to some other kinds of planning algorithms.

This, of course, leaves open the question of which search algorithm should be used in order to find a goal state effectively and efficiently within the forward-chaining search tree.

At first, one might consider using standard search algorithms such as breadth first or iterative deepening. But if the shortest plan consists of 100 actions, and the branching factor is 1000 (that is, there are 1000 applicable actions at each step), breadth first search would have to investigate $1000^{100} = 10^{300}$ nodes – and more importantly, would have to *store* most of those nodes in memory, which would clearly be impossible. Using iterative deepening would require less memory, but would still require too much time. Even at 10^{12} nodes per second (which will not be realistic for many years to come, since it corresponds to testing 300 nodes per clock cycle on today's fastest processors), finding a plan using pure iterative deepening could require 10^{280} years, though it could be found earlier if an optimal or near-optimal plan would happen to be found in the part of the tree that was searched first. It is clear that using these algorithms would lead to very inefficient planners, due to the combinatorial explosion in the number of potential plans when the size of a planning problem increases. But this problem is not inherent in the concept of forward-chaining: It is caused by using "blind" brute force search algorithms without any form of goal-directed behavior.

One can identify two main categories of techniques being used for goal-directed search in recent forward-chaining planners.

Some planners use various forms of *heuristics* to guide the search process. The heuristic function helps determine which nodes and branches should be investigated before others, and can either be automatically extracted from the planning domain and problem instance or be specified explicitly by the user. Since the heuristic function is generally not perfect, occasionally misjudging which branch would lead to the goal more quickly, nodes that are less preferred by the heuristic function are saved so that the planner can return to them at a later stage. HSP (Bonet & Geffner, 1998) and FF (Hoffmann & Nebel, 2001) are two very successful planners that automatically determine a heuristic function given a planning domain and a problem instance.

Other planners use domain-independent or domain-dependent *pruning*, actually removing search nodes completely from the tree. Cycle checking is a simple form of domain-independent pruning, where a node is pruned if it visits a state that has already been visited on the path from the root node. In the blocks world, picking up a block, stacking it on another block, unstacking it again and returning it to the table would lead to such a state cycle. Any plan containing these four actions would still be executable if the actions were to be removed. The final state generated by the plan would be identical, so as long as the goal only constrains the final state, as opposed to placing constraints on the path taken before that state is reached, it would be satisfied in the abbreviated plan iff it was satisfied in the original plan.

As will be described in the next section, Blockhead and TLPlan are two planners that prune the search tree using domain-dependent pruning techniques.

5.3 Blockhead, TLPlan, and Control Formulas

All the different types of information discussed in Section 5.1 are more or less required in order to specify the planning problem to be solved. For example, in order to determine in advance whether an operator sequence will be executable, a planner needs information about operators and about the initial state where plan execution will begin, and in order to determine whether the goal is achieved, complete information about the acceptable goal states is also needed. Planners which only depend on this type of information are often called *knowledge-sparse* or *fully automated* planners.

Some planners also accept additional information which is in some sense redundant but which may help improve the performance of the planner or the quality of the solution plan. In the PRODIGY planner, for example, it is possible to specify rules that specify which subgoals should be solved before other subgoals, or rules that determine which operator arguments should be preferred over others (Carbonell, Blythe, Etzioni, Gil, Joseph, Kahn, Knoblock, Minton, Pérez, Reilly, Veloso, & Wang, 1992; Veloso, Carbonell, Pérez, Borrajo, Fink, & Blythe, 1995). Planners using heuristic search methods may allow the user to specify heuristic evaluation functions explicitly rather than using a built-in function. The input to a hierarchical task network (HTN) planner contains a set of task decomposition rules that determine how the main objective ("goal") is decomposed into primitive actions (Sacerdoti, 1975; Tate, 1977; Vere, 1983; Wilkins, 1988; Currie & Tate, 1991; Erol, Hendler, & Nau, 1994; Nau, Cao, Lotem, & Muños-Avila, 2001; Nau, Au, Ilghami, Kuter, Murdock, Wo, & Yaman, 2003). In addition to specifying true constraints inherent in the domain being modeled, such rules can also be used to provide a significant measure of additional information to the planner

Planners that accept additional information may be completely dependent upon this information or may have reasonable performance even with a pure problem specification. In either case, they are sometimes called *knowledge-based* or *hand-tailored* planners. They have also occasionally been called domain-dependent planners, although it could be argued that this is a misnomer, because just like knowledge-sparse planners, knowledge-based planners are generally applicable to arbitrary domains as long as the proper domain specification is provided as part of the input. What is truly domain-dependent is not the planner but the domain specification – but then again, even knowledge-sparse planners require domain specifications, and even those domain specifications must necessarily contain domain-specific information such as operator definitions. Knowledge-based planners may require additional types of information, but this is merely a difference in degree, not a difference in kind.

It should be emphasized that just as there is no binary distinction between domain-dependent and domain-independent planners, there is no binary distinction between hand-tailored and fully automated planners. Many "fully automated" planners perform considerably better given some tuning for each specific domain

– but this tuning is hidden in command line arguments rather than being openly specified in a domain description. Even those planners that have no such command line arguments could potentially be made even more fully automated if they could automatically determine some information about operators or the initial state from their execution environment. In addition, some "hand-tailored" planners such as PbR (Ambite, 1998; Ambite, Knoblock, & Minton, 2000; Ambite & Knoblock, 2001) can to some extent generate the additional information themselves in a preprocessing step, further blurring the boundary between the two kinds of planners.

5.3.1 Blockhead: A Domain-Dependent Planner

Consider once more the blocks world search tree in Figure 5.5 on page 130. Cycle checking has already pruned some nodes from this tree, but most of the tree still remains. If we consider the tree more carefully, though, we can see a few operator sequences that are obviously bad, sequences that the planner could potentially detect in order to backtrack and try a more promising approach. For example, the plan $\langle \mathsf{unstack}(\mathbf{C}, \mathbf{B}), \mathsf{stack}(\mathbf{C}, \mathbf{E}) \rangle$ would place **C** on top of block **E**. If the goal requires **C** to be on top of **E**, then this might initially appear to be a good move. But **E** is not in its final location. It will eventually have to be moved to its final location, and if there are blocks on top of it, those blocks will first have to be removed. In this situation, it would be a good idea to forbid the planner from placing **C** on top of **E**, forcing it to prune this search node, backtrack, and try other solutions such as placing **C** on the table instead. This will always be a viable alternative, given that the table in the standard blocks world never runs out of space.

This principle, which can also be stated more succinctly as "only add blocks to good towers", is one of the four rules that were used in Blockhead, a domain-specific blocks world planner by Kibler and Morris (1981). Each of the four rules pruned branches corresponding to actions that definitely would not take the planner closer to the goal, actions where there were definitely better alternatives available – as the authors expressed it, "don't be stupid".

Though no precise benchmark results were published, the authors reported that the techniques used in Blockhead led to significant reductions in the number of nodes visited by the planner and thereby significant performance improvements.

5.3.2 TLPlan: A Hand-Tailored Planner

Blockhead was truly a domain-dependent planner, written for the blocks world. TLPlan, on the other hand, is a domain-independent planner with an extended domain description language permitting the user to write explicit domain-specific *control rules* that help control the search process, determining which nodes should or should not be pruned (Bacchus & Kabanza, 2000).

Control rules for a forward-chaining planner could potentially take many different shapes. In TLPlan, control rules are formulas expressed using a variation of the modal tense logic LTL, Linear Temporal Logic (Emerson, 1990). We will use the terms *control rule* and *control formula* interchangeably.

LTL is defined by taking a standard first-order language and adding four temporal modalities: U (until), $\Diamond$ (eventually), $\Box$ (always), and $\bigcirc$ (next). Formulas in LTL are interpreted over *timelines*, sequences $\langle s_0, s_1, \ldots \rangle$ of *states* (called *worlds* in TLPlan terminology), where each state is an interpretation for the original first-order language. The semantics of an LTL formula is defined relative to a *current state*, a position within the sequence of states. Intuitively, $\bigcirc \alpha$ means that α holds in the next state along the timeline, $\Box \alpha$ means that α holds in the current state and all future states, $\Diamond \alpha$ means that α holds in the current state or some future state, and $\alpha \mathsf{U} \beta$ means that β will eventually hold, and for the entire interval of time until it does, α holds. See Emerson (1990) or Bacchus and Kabanza (2000) for a precise definition of LTL.

Bacchus and Kabanza add to LTL a *goal modality*, goal, where the formula $\mathsf{goal}(\phi)$ is true iff ϕ holds in every acceptable goal state, or equivalently, iff the goal entails ϕ. This additional modality is the key to enabling control rules to provide a measure of goal-directed guidance to the planner. Since the TLPlan implementation is restricted to using conjunctive goals, entailment checks can be made efficiently and do not require theorem proving.

To ensure that quantified formulas can be evaluated by iteration over a finite set of values, Bacchus and Kabanza use bounded quantifiers of the form $\forall[x : \alpha(x)].\phi$ and $\exists[x : \alpha(x)].\phi$, where α is a positive literal that can only hold for finitely many value constants and where the quantified formula ϕ only has to be evaluated for those x that satisfy $\alpha(x)$. This also improves performance: Rather than letting the planner iterate over all value constants mentioned in a problem instance, or forcing it to automatically choose a literal from which to extract a set of values to iterate over, the user can manually specify a literal likely to hold only for a small set of values.

Example 5.3.1 (Blocks World, continued)
If a block is placed on top of the destination of another block, then this action must eventually be undone in order to reach the goal configuration. Such actions may be necessary in a bounded blocks problem, where the table does not have sufficient space for all blocks, but they are never required in the standard blocks world.

In other words, it is always the case that if you are currently holding a block x, which might potentially be placed on top of some clear block y, and there is a goal that z should be on top of y, and x is not in fact the same block as z, then at the next timepoint you should not have placed x on top of y.

$$\Box\ \forall[x : \mathsf{holding}(x)]\forall[y : \mathsf{clear}(y)]\forall[z : \mathsf{goal}(\mathsf{on}(z,y))].x \neq z \rightarrow \bigcirc\neg\mathsf{on}(x,y) \qquad \blacksquare$$

Example 5.3.2 (Blocks World, continued)
Assume the existence of a defined predicate goodtower(b) which holds iff b is the topmost block in a tower which does not violate the goal, that is, a tower in which

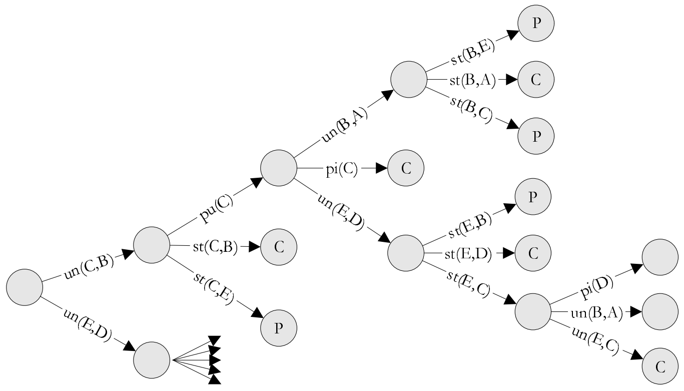


Figure 5.6: Pruning a Forward Chaining Search Space

no blocks will have to be moved. Then the Blockhead rule "only add blocks to good towers" can be succinctly represented as follows:

$$\Box\ \forall[x : \mathsf{clear}(x)]\ \neg\mathsf{goodtower}(x) \rightarrow \bigcirc\neg\exists[y : \mathsf{on}(y, x)]\ \mathtt{true}$$

(The consequent of the implication should state that there is no block y on top of x, but because of the use of bounded quantification, this condition becomes integrated into the bound. Only `true` remains to the right of the existential quantifier.) ■

These two simple rules reduce the search space considerably. Though this effect is far more visible for larger problem instances with greater depth and higher branching factors, it can also be seen for smaller problem instance, as in Figure 5.6 which shows a pruned version of the search space originally shown in Figure 5.5. The node label "P" indicates a node that has been pruned due to a control rule.

5.3.3 Testing Control Formulas Using Progression

Though we have stated that control formulas help the planner determine which nodes should be pruned, we have up to this point been somewhat vague regarding the exact manner in which this is done.

Control rules are intended to improve the goal-directedness of a forward-chaining planner, and therefore its performance, by allowing parts of the search space to be pruned. This naturally cannot be achieved by waiting until a plan has been found that satisfies the classical goal and then testing whether the plan also satisfies all control rules. Instead, it requires a condition that can be tested in intermediate nodes in the search tree.

One potential approach would be to prune every intermediate node that does not satisfy all control rules. This may initially seem quite reasonable, until the realization dawns that some control formulas that are violated in a certain search node can be satisfied in its children. To see how this can be the case, we appeal to yet another control formula for the blocks world.

Example 5.3.3 (Blocks World, continued)
The blocks world control rule from Example 5.3.1 is applicable when holding a block, and places certain restrictions on the locations where this block can be put down. In some cases it is possible to strengthen this constraint. For example, if the crane is currently holding a block whose final destination is clear (nothing is on top of it) and is a good tower (will not have to be removed), then clearly this is where the block should be put down.

$$\Box\, \forall[x{:}\mathsf{holding}(x)]\forall[y{:}\mathsf{goal}(\mathsf{on}(x,y))]\ \ \mathsf{clear}(y) \wedge \mathsf{goodtower}(y) \rightarrow \bigcirc\, \mathsf{on}(x,y) \qquad \blacksquare$$

Unlike the first two control formulas, which stated what *must not* happen, this formula places requirements on changes that *must* take place in the world – under certain circumstances, you must place block x on top of block y. If this formula were to be evaluated in a state sequence where block x had just been picked up, then the formula would be violated, simply because the action stacking the block on top of its destination had not yet been added to the plan.

A more reasonable semantics would state that all control formulas should hold in the final solution. Without the need to consider what might hold in the prefixes of a solution, it is far less likely that a domain designer would inadvertently write a control rule that prunes branches that could have led to good solutions. Inner nodes can still be pruned from the search tree as long as it can be proven that there is a control formula that must be violated in all of its descendants.

TLPlan uses a formula progression algorithm to achieve a similar semantics, with a slight difference: Control formulas are only viewed as guidance, not as goals, and do not necessarily need to be satisfied in the final solution. This difference is usually not important for classical planning domains, but care should be taken not to use TLPlan control formulas as a means of implementing temporally extended goals such as safety goals. For example, the control formula $\Diamond\, \phi$ can never be violated, because the progression algorithm always assumes ϕ might be satisfied in some future state – even if $\phi = \mathtt{false}$.[4]

Consider a finite or infinite sequence of states $\langle s_0, s_1, s_2, \ldots \rangle$ and a modal tense logic formula ϕ that should hold in this sequence. Progressing this formula through the state s_0 yields a formula ϕ^+ that should hold in the remainder of the state sequence: ϕ holds in $\langle s_0, s_1, \ldots \rangle$ iff ϕ^+ holds in $\langle s_1, s_2, \ldots \rangle$. If the information in s_0 was sufficient to prove that the formula must be false, regardless of the what comes after s_0 in the timeline, the progression algorithm may immediately return `false`.

In the following, $\otimes$ denotes a binary logical connective, and the expression $\phi[x \mapsto c]$ denotes the formula resulting from replacing all free occurrences of x in the formula ϕ with the constant c.

[4]The TLPlan implementation also allows the specification of temporally extended goals (Bacchus & Kabanza, 1996a, 1998), modal temporal formulas that must be satisfied by the solution plan. However, this cannot be used in combination with state-based goals, and also prevents the use of the goal modality.

Definition 5.3.1 (Progression for LTL Formulas)

```
1   procedure Progress($\phi, s$)
2   if $\phi$ contains no temporal modalities
3       if $s \models \phi$ return true else return false
4   if $\phi = \phi_1 \otimes \phi_2$ return Progress($\phi_1, s$) $\otimes$ Progress($\phi_2, s$)
5   if $\phi = \neg\phi_1$ return $\neg$Progress($\phi_1, s$)
6   if $\phi = \bigcirc\, \phi_1$ return $\phi_1$
7   if $\phi = \phi_1 \mathsf{U}\, \phi_2$ return Progress($\phi_2, s$) $\vee$ (Progress($\phi_1, s$) $\wedge\, \phi$)
8   if $\phi = \Diamond\, \phi_1$ return Progress($\phi_1, s$) $\vee\, \phi$
9   if $\phi = \Box\, \phi_1$ return Progress($\phi_1, s$) $\wedge\, \phi$
10  if $\phi = \forall[x : \gamma(x)]\phi$ return $\bigwedge_{c:s\models\gamma(c)}$ Progress($\phi[x \mapsto c], s$)
11  if $\phi = \exists[x : \gamma(x)]\phi$ return $\bigvee_{c:s\models\gamma(c)}$ Progress($\phi[x \mapsto c], s$)
```

Unnecessary occurrences of the truth constants `true` and `false` are removed from the result using the standard rules $\neg$`false` = `true`, (`false` $\wedge\, \alpha$) = ($\alpha \wedge$ `false`) = `false`, (`false` $\vee\, \alpha$) = ($\alpha \vee$ `false`) = α, $\neg$`true` = `false`, (`true` $\wedge\, \alpha$) = ($\alpha \wedge$ `true`) = α, and (`true` $\vee\, \alpha$) = ($\alpha \vee$ `true`) = `true`. ■

Recall that each node in the forward-chaining search tree corresponds to a plan, and that each plan is associated with a state sequence that would be generated if the plan were to be executed starting in the initial state. TLPlan introduces one additional item of information for each node: A control formula, which has been progressed through the state sequence associated with the node.

The initial node corresponds to the empty plan, and its associated state sequence only contains one state: The initial state, as specified by the user. No further states have yet been created. Consequently, the control formula associated with this node is generated by conjoining the initial control formulas specified by the user and progressing the resulting conjunction through the initial state.

Applying an operator instance to an existing node n in a sequential forward-chaining search tree always yields one or more new states at the end of the state sequence for n. The control formula associated with n is progressed through the new states, resolving all references to fluents in those states, and the new node being generated is labeled with the progressed formula.

If progression returns the formula `false` at any point, the planner can immediately backtrack, knowing that the formula is violated in all possible extensions to the current plan candidate.

In addition to providing a reasonable semantics for control rules, this method also ensures that control formulas never have to be evaluated in the same state twice, which is very important for the performance of TLPlan. On the other hand, a potentially large progressed control formula must be stored in each open search node, which can considerably increase the memory consumption of the planner.

As demonstrated by Bacchus and Kabanza (2000), this technique led to some very impressive improvements in efficiency for many well-known benchmark domains when compared to planners such as Blackbox (Kautz & Selman, 1998, 1999) and IPP (Koehler et al., 1997), two of the leading competitors in the First International Planning Competition (IPC-1998, held at the AIPS-1998 conference: McDermott, 1998).

Chapter 6

TALplanner

This chapter introduces TALplanner, a new planner which is inspired by the use of temporal control formulas in TLPlan (Bacchus & Kabanza, 2000). TALplanner has some similarities to TLPlan, but also some significant differences, the most obvious one being the use of a TAL-based logic which has given the planner its name. Some of the most important similarities and differences between the two planners will be covered in the following section, together with a brief overview of the structure of the planner and its use of TAL. The remaining sections in this chapter will go into considerably more detail regarding the way planning domains and control formulas are modeled in TAL and the extensions that have been made to TAL in order to model certain planning-specific concepts. A set of preliminary benchmark comparisons for an early version of TALplanner will also be presented.

6.1 An Overview of TALplanner

As mentioned in the introduction to this thesis, one of the many design goals for TALplanner was the intention to develop a planner using the TAL semantics for actions and world descriptions. The TAL logics are specifically developed for reasoning about action and change, and given a few planning-related extensions, one logic in particular, TAL-C (Chapter 2; Karlsson & Gustafsson, 1999), provides a suitable framework for modeling complex, potentially concurrent actions as well as most other concepts required to succinctly capture the essential features of a planning domain (Section 6.2).

Using TAL in this manner provides a declarative first-order semantics for planning domains, an important difference from TLPlan where only control formulas are based on the use of logic and actions are instead modeled using an operational semantics. But unlike Green's approach (1969), which involved not only representing planning domains in logic but also *generating* plans using a resolution theorem prover, the declarative semantics of TAL serves mainly as a specification for the proper behavior of the planning algorithm. The TALplanner implementation generates plans using standard procedural forward-chaining search methods (Section 6.3) together with a search tree which is pruned with the help of temporal control formulas.

Though the use of control formulas is inspired by TLPlan, there are extensive differences in how the two planners test whether a plan satisfies or violates a given control formula. These differences were initially prompted by the different properties of Linear Temporal Logic (LTL), used for control formulas in TLPlan, and TAL, which should preferably be used for control formulas in TALplanner in order to avoid the need to mix two logics in one planner. Whereas LTL is a modal temporal logic with tense operators that can be used together with a progression algorithm, TAL uses explicit time, which is less suitable for progression. TAL-based control formulas are therefore tested in a different manner, involving a pre-processing phase generating formulas suitable for incremental evaluation in successive plan candidates (Section 6.4). The formulas yielded by this pre-processing are amenable to further analysis and optimizations, which has in turn considerably improved the performance of TALplanner (Chapter 8). Because of the desire to compare the two types of control formulas within the same implementation, and because each turns out to have the potential for higher performance for different classes of formulas, tense formulas can also be emulated within TAL and used together with a progression algorithm (Section 6.5). The current version of TALplanner integrates the separate progression-based and evaluation-based algorithms presented in early papers (Doherty & Kvarnström, 1999; Kvarnström et al., 2000), allowing both types of control formulas to be used in the same planning domain.

Extensions to TAL-C. The high-level $\mathcal{L}(\mathrm{ND})$ language for the TAL-C logic must be extended somewhat in order to allow the specification of goals and control formulas. Additional extensions have been introduced in order to provide more succinct representations of certain planning-related concepts even though they could in fact be modeled in pure TAL-C. Eventually, the extended logic may become a new planning-specific TAL logic named TAL-P. For the moment, however, changes are still being made to the logic, and the intermediate version is viewed only as a temporary extension to the macro language $\mathcal{L}(\mathrm{ND})$ from TAL-C. The extended version of $\mathcal{L}(\mathrm{ND})$ is denoted by $\mathcal{L}(\mathrm{ND})^*$.

As for several other extensions to the TAL logics, the intention behind $\mathcal{L}(\mathrm{ND})^*$ has been to avoid changing the standard first-order TAL base logic $\mathcal{L}(\mathrm{FL})$, each new addition being accompanied with an extension to the $\mathit{Trans}()$ translation function translating $\mathcal{L}(\mathrm{ND})^*$ formulas into $\mathcal{L}(\mathrm{FL})$ (Section 2.4.1). In a sense, this has been an interesting research topic in itself: To what extent can TAL be adapted to the planning task without modifications to the original $\mathcal{L}(\mathrm{FL})$ language? As will be seen, there are concepts that could have been modeled in a more elegant manner had this restriction been lifted. On the other hand, using the same base logic across

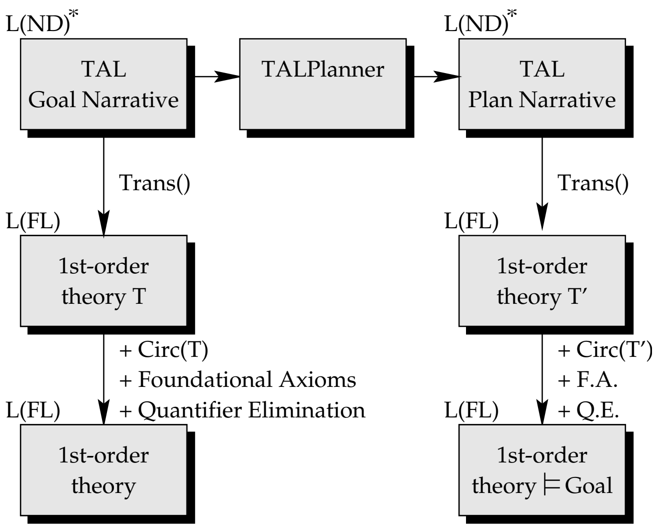


Figure 6.1: The relation between TAL and TALplanner

multiple versions of TAL also has advantages in terms of facilitating comparisons between TAL logics as well as retaining existing results from research based on the standard version of $\mathcal{L}(\mathrm{FL})$.

Expressivity. Though TAL-C provides an expressive semantics for domain descriptions, it is not necessarily a good idea to introduce all of this expressivity into TALplanner at once. Instead, it would be better to use a smaller subset of TAL-C in the first version of the planner, and then to incrementally introduce higher levels of expressivity in a series of well-defined extensions to the basic TALplanner algorithms. A number of such extensions have been made since the first version of TALplanner. Most of these extensions have already been integrated into the current description of the modeling language as specified in Section 6.2. The main exception to this is the support for concurrency, which will be discussed separately in Chapter 7.

Planning as Narrative Generation. Due to the narrative-based nature of TAL, it may be useful to view planning as a form of narrative generation. Figure 6.1 contains an extended version of the diagram previously shown in Figure 2.2 on page 23. As seen in the top row of this figure, the input to TALplanner is a narrative in the extended macro language $\mathcal{L}(\mathrm{ND})^*$. This narrative is sometimes called a *goal narrative*, emphasizing the fact that it specifies a planning problem instance, and is usually denoted by $\mathcal{N}$. The goal narrative consists of two parts: A domain description, defining among other things the operators that are available to the planner, and a problem instance description, defining the initial state and the goal. TALplanner uses this high-level description of a planning problem to search for a set of TAL-C action occurrences (plan steps) that can be added to this narrative so that in the corresponding logical model, a goal state is reached. If this succeeds, the output is a new TAL narrative in $\mathcal{L}(\mathrm{ND})^*$ where the appropriate set of TAL action occurrences has been added. This narrative is sometimes called a *plan narrative*, emphasizing the fact that it represents a solution to a planning problem. Both goal narratives and plan narratives can be translated into $\mathcal{L}(\mathrm{FL})$ (the second row in the figure). As in pure TAL-C, a number of foundational axioms are required, and a standard TAL circumscription policy is applied, yielding complete definitions of the *Occlude* and *Occurs* predicates (the third row). Further details will be presented in Section 6.2.10.

6.2 Representing Planning Problems in TAL

This section introduces $\mathcal{L}(\mathrm{ND})^*$, a new TAL-based macro language for use in applications related to planning. Many concepts and definitions used by $\mathcal{L}(\mathrm{ND})^*$ are inherited from the TAL-C version of $\mathcal{L}(\mathrm{ND})$ as specified in Chapter 2, though there are also a number of new formula classes and statement classes as well as a number of restrictions compared to $\mathcal{L}(\mathrm{ND})$. To better show where TALplanner and $\mathcal{L}(\mathrm{ND})^*$ fall in the spectrum of expressivity for domain modeling, the section is loosely based on the same structure as Section 5.1.

Notational Conventions. All formulas and $\mathcal{L}(\mathrm{ND})$ statements in the following chapters will be shown using the input syntax for TALplanner, with the exception of some connectives and quantifiers that may be written using the ordinary logical symbols for increased clarity.

Variables are typed. Value variables are usually given the same name as the corresponding sort written in italics, possibly with a prime and/or an index. For example, *block*, *block′* and *block*$_3$ would be variables of sort `block`. Similarly, variables named t or τ are normally temporal variables, and variables named n are normally integer-valued variables. All free variables are implicitly universally quantified.

6.2.1 Single Timepoint Formulas and Terms

In the following, there will sometimes be a need for formulas or terms that only reference fluent values at a single specific timepoint. This requires restrictions on the two different ways of specifying a temporal context for a fluent term: The temporal context prefix $[\tau]$ and the fluent value function $\mathit{value}(\tau, f)$.

Definition 6.2.1 (Single Timepoint Formula)
A *single timepoint formula* for the timepoint τ is a static formula where all fixed fluent formulas are of the form $[\tau]\ \alpha$ (where α is a fluent formula) and all occurrences of the *value* function are of the form $\mathit{value}(\tau, f)$ (where f is a fluent term). ■

Definition 6.2.2 (Single Timepoint Value Term)
A *single timepoint value term* for the timepoint τ is a value term where all occurrences of the *value* function are of the form $\mathit{value}(\tau, f)$ (where f is a fluent term). ■

Definition 6.2.3 (Single Timepoint Fluent Term)
A *single timepoint fluent term* for the timepoint τ is a fluent term where all occurrences of the *value* function have the form $\mathit{value}(\tau, f)$ (where f is a fluent term). ■

6.2.2 Describing the World: States, State Variables, and Objects

As in most planners, we assume that the current state of the world can be described in terms of a finite set of typed state variables, each of which takes a fixed number of typed arguments. These state variables can easily be represented as TAL fluents. Except for defined fluents (Section 6.2.3), all fluents are implicitly declared persistent.[1]

TAL uses an order-sorted type system, and can therefore directly represent a hierarchical type structure rather than representing types as unary predicates or as a flat set of types. There is a standard sort `boolean` $= \{\textbf{true}, \textbf{false}\}$. Additional sorts can be defined by the user. As explained in Chapter 2, type information has generally been specified informally in the description of a TAL narrative, and we will continue to use the VITAL and TALplanner input syntax to describe these parts of the vocabulary, with an extension to allow TAL value domain specifications to be split into domain declarations (placed in the planning domain specification) and object definitions (placed in each problem instance specification).

TALplanner also allows the use of numeric types. In order to keep the semantics of these types clear, only integers and fixed point numbers (that is, numbers with a fixed number of decimals) are allowed, and lower and upper bounds must be declared for each numeric type. All of the standard arithmetic operators are available for the numeric types and are given an interpretation through semantic attachment.

Example 6.2.1 (Logistics Domain, continued)
The logistics domain was first defined in Example 5.1.3 on page 119. A hierarchy of seven types is defined for the entities present in the logistics domain: The type `loc` (location) has the subtype `airport`. All locations are in a `city`. The type `thing` has the subtypes `obj` and `vehicle`, the latter of which has the subtypes `truck` and `plane`. Types are defined in terms of *value domains* and are therefore declared using the label **domain**, not to be confused with a planning domain.

There are two boolean fluents, at(`thing`,`loc`) and in(`obj`,`vehicle`). The city-valued fluent city-of demonstrates the use of non-boolean state variables: At any timepoint, city-of$(\mathit{loc}) \hat{=} \mathit{city}$ means that the location *loc* is in the city *city*.

[1]Though there would be no intrinsic problem in allowing the use of durational fluents, existing planning domains only use persistent fluents. Limiting the planner to using persistent fluents streamlines the presentation of certain concepts.

```
domain    city, loc, thing
domain    airport :parent loc
domain    obj, vehicle :parent thing
domain    truck, plane :parent vehicle

feature   at(thing,loc) :domain boolean
feature   in(obj,vehicle) :domain boolean
feature   city-of(loc) :domain city
```

The logistics problem instance in Figure 5.3 on page 120 uses the following objects:

```
objects   :domain loc :elements { city1-1, city1-2, city2-1, city2-2,
                                  city3-1, city3-2, city4-1, city4-2 }
objects   :domain airport :elements { city1-2, city2-2, city3-2, city4-2 }
objects   :domain city :elements { city1, city2, city3, city4 }
objects   :domain obj :elements { package-1, package-2, package-3,
                                  package-4, package-5, package-6 }
objects   :domain truck :elements { truck-1, truck-2, truck-3, truck-4 }
objects   :domain plane :elements { plane-1 }
```
■

Example 6.2.2 (Gripper Domain, continued)
The gripper domain was first defined in Example 5.1.1 on page 118. This domain uses the value domains `obj` for objects (including the robot **robby**), `ball` for balls (a subtype of `obj`), `room` for rooms, and `gripper` for grippers. The fluent loc specifies the location of objects, free specifies whether a given gripper is free, and is-carried-in specifies whether the robot is carrying a certain object in a certain gripper.

```
domain    obj
domain    ball :parent obj
domain    room, gripper
feature   loc(obj) :domain room
feature   free(gripper) :domain boolean
feature   is-carried-in(ball, gripper) :domain boolean
```

The gripper problem instance in Figure 5.1 on page 118 uses the following objects:

```
objects   :domain obj :elements { robby }
objects   :domain ball :elements { ball1, ball2 }
objects   :domain room :elements { roomA, roomB }
objects   :domain gripper :elements { left, right }
```

Objects declared to belong to a particular value domain are automatically also members of all ancestor domains. Thus, `obj` will contain three objects: **robby**, which is explicitly declared to belong to the domain, and **ball1** and **ball2**, which are members of `obj` by virtue of being declared to belong to the subdomain `ball`.

■

Example 6.2.3 (Blocks World, continued)
In the blocks world planning domain, first defined in Example 5.1.2 on page 118, there is only one value domain: block, containing a set of blocks. There are five fluents: on$(block_1, block_2)$ holds if $block_1$ is on top of $block_2$, while ontable$(block)$ holds if *block* is on the table. If a block is neither on the table nor on top of another block, we must be holding it, which is modeled using the fluent holding$(block)$. A block is clear iff it is possible to place another block on top of it, which is modeled using the fluent clear$(block)$. Finally, the hand is empty iff it is not holding a block, which is modeled using the fluent handempty. Following the standard formulation, all fluents in this domain are boolean.

```
domain   block
feature  on(block, block) :domain boolean
feature  clear(block), ontable(block) :domain boolean
feature  holding(block), handempty :domain boolean
```

The example problem instance in Figure 5.2 uses the following objects:

```
objects  :domain block :elements { A, B, C, D, E }
```

■

6.2.3 Defined Predicates and Fluents

As noted in the introduction to planning, it often makes sense to define concepts in a planning domain hierarchically, specifying a set of primary predicates which are directly affected by actions as well as a set of secondary predicates which are defined in terms of formulas. TALplanner is not limited to boolean predicates and therefore supports the more general concept of defined (possibly non-boolean) *fluents*.

Each defined fluent is declared in the narrative background specification. The TALplanner fluent declaration syntax is extended with a new flag, :defined, for this purpose. The definition of a defined fluent should be specified in terms of a fluent definition statement, also in the narrative background specification.

Definition 6.2.4 (Fluent Definition Statement)
A *fluent definition statement*, labeled define, is one of the following:

- A labeled statement **define** $[t]\ f(v_1, \ldots, v_n) : \phi$, where t is a temporal variable, f is a boolean feature name, v_1 through v_n are distinct value variables of sorts corresponding to the argument sorts of f, and ϕ is a single timepoint formula for t where t and $v_1, \ldots, v_n$ may occur as free variables.

- A labeled statement **define** $[t]\ f(v_1, \ldots, v_n) : \omega$, where t is a temporal variable, f is a non-boolean feature name, v_1 through v_n are distinct value variables of sorts corresponding to the argument sorts of f, and ω is a single timepoint value term for t where t and $v_1, \ldots, v_n$ may occur as free variables.

■

Example 6.2.4 (Gripper Domain, continued)
In the gripper domain, it would be possible to create a boolean defined fluent is-carried(`ball`) which is true exactly when there exists a gripper carrying the given ball, that is, exactly when $\exists gripper[\text{is-carried-in}(ball, gripper)]$. This example can be modeled as follows:

feature is-carried(`ball`) :domain `boolean` :defined
define $[t]$ is-carried$(ball)$: $[t]\ \exists gripper\ [\ \text{is-carried-in}(ball, gripper)\ ]$ ■

The Semantics of Defined Fluents

Ordinary fluents are given a unique value in the initial state. The inertia assumption is applied, ensuring that each fluent retains the value from the previous timepoint except at those timepoints where an action explicitly assigns a new value.

Defined fluents, on the other hand, should have the same value as a formula or as a value term. They are therefore implicitly declared dynamic, implying that they are not constrained by an inertia assumption or a default value assumption. This leaves the task of ensuring that each defined fluent takes on the intended value at all points in time. Without loss of generality, the remainder of this discussion will mainly be restricted to the case of boolean defined fluents, defined in terms of a logic formula.

In the simplest case, a fluent definition formula may only refer to primary (non-defined fluents). This is the case in the gripper example above, where the definition formula only refers to the is-carried-in fluent. In this case, finding the value of a fluent instance is quite easy. In terms of an operational semantics, the planner should simply evaluate the definition formula. In terms of a declarative semantics, the fluent should take on the value **true** iff the definition formula holds, which can easily be achieved by translating each fluent definition statement into a TAL domain constraint. Given that the primary fluents on which the definition depends already have definite values, this is guaranteed to provide a unique value to each instance of the defined fluent at any point in time.

Definition 6.2.5 (Translation of Fluent Definition Statements)
Fluent definition statements are translated into $\mathcal{L}(\text{FL})$ as follows:

- $Trans([t]f(v_1, \ldots, v_n) : \phi) = Trans(\forall t, v_1, \ldots, v_n[[t]f(v_1, \ldots, v_n) \leftrightarrow \phi])$
- $Trans([t]f(v_1, \ldots, v_n) : \omega) = Trans(\forall t, v_1, \ldots, v_n[[t]f(v_1, \ldots, v_n) \hat{=} \omega])$ ■

Example 6.2.5 (Gripper Domain, continued)
The fluent definition statement in Example 6.2.4 can be translated into the following $\mathcal{L}(\text{FL})$ domain constraint:

dom $\forall t, ball\ [\ Holds(t, \text{is-carried}(ball), \mathbf{true}) \leftrightarrow$
$\exists gripper\ [\ Holds(t,\ \text{is-carried-in}(ball, gripper), \mathbf{true})\ ]]$ ■

If fluent definition formulas may refer to defined fluents, but cannot refer back to the fluent being defined (either directly or indirectly through the definitions of other defined fluents), the value of a fluent instance can be found in the same manner. In operational terms, evaluating such a fluent definition formula may lead to further calls to the formula evaluator for nested defined fluents, but such calls will eventually be grounded in the evaluation of primary fluents.

Difficulties only arise when a fluent definition formula may directly or indirectly refer back to the fluent instance being defined. In terms of a declarative first-order semantics, such recursive definitions may lead to unintended models. For example, at first glance it might appear as if a recursively defined fluent could be used to define the transitive closure of the on fluent in the blocks world:

$$\textbf{define}\quad [t]\ \mathsf{above}(x,y)\ :\ [t]\ \mathsf{on}(x,y)\ \vee\ \exists z\,[\,\mathsf{on}(x,z)\ \wedge\ \mathsf{above}(z,y)\,]$$

Unfortunately, it is well known that the transitive closure of a relation cannot be defined in first order logic, which means that our first-order translation of this formula, as defined above, cannot provide the intended definition of above.

This can of course be "solved" by using a more powerful semantics. The planning domain definition language PDDL2.2 (Edelkamp & Hoffmann, 2004) uses a semantics where predicate definitions are viewed as derivation rules where a defined predicate holds iff it can be derived using successive applications of these derivation rules, with syntactic restrictions to ensure that the order in which rules are applied is unimportant. This addition to PDDL2.1 was made explicitly in order to permit the modeling of transitive closures. TLPlan (Bacchus & Kabanza, 2000) uses an operational semantics based on recursive formula evaluation which also allows the modeling of transitive closures. For example, evaluating the fluent $\mathsf{above}(\mathbf{A},\mathbf{B})$ leads to a recursive call to the formula evaluator for the formula $\mathsf{on}(\mathbf{A},\mathbf{B})\ \vee\ \exists z\ [\ \mathsf{on}(\mathbf{A},z)\ \wedge\ \mathsf{above}(z,\mathbf{B})\ ]$.

The TALplanner implementation also supports recursively defined fluents, with a semantics identical to that of TLPlan. A new version of TAL-C will eventually be developed based on a fixpoint base logic (Immerman, 1998) rather than a first-order base logic, at which time there will also be a logic-based semantics for recursively defined fluents.

6.2.4 The Beginning of Time: The Initial State

TALplanner currently requires complete information about all fluents in the initial state (time 0). This information is specified in terms of *initialization statements* providing facts known to hold at time 0.

The planner is not limited to plain observations of facts, but permits the use of arbitrary fluent formulas, including existential quantification and disjunction, as long as these formulas define a unique initial state. Compared to planners that use a simple list of initial facts together with a closed world assumption (CWA) where all facts that are not explicitly listed are assumed to be false, this representation is less compact but also more flexible. (For convenience, the TALplanner implementation also allows a variation of CWA initialization.)

Definition 6.2.6 (Initialization Statement)
An *initialization statement*, labeled **init**, is a single timepoint formula for time 0. ■

Example 6.2.6 (Gripper Domain, continued)
The following example shows the specification of the initial state for the gripper problem instance in Figure 5.1 on page 118. In this initial state, all grippers are free and the robot and all balls are in room A.

$\textbf{init}\ \forall \mathit{gripper}, \mathit{ball}\ [\ [0]\ \mathsf{free}(\mathit{gripper}) \land \neg\mathsf{is\text{-}carried\text{-}in}(\mathit{ball}, \mathit{gripper})\ ]$
$\textbf{init}\ \forall \mathit{obj}\ [\ [0]\ \mathsf{loc}(\mathit{obj}) \mathrel{\hat{=}} \textbf{roomA}\ ]$ ■

Example 6.2.7 (Logistics Domain, continued)
The initial state of the logistics problem instance in Figure 5.3 on page 120 can be defined as follows:

$\textbf{init}\ [0]\ \mathsf{city\text{-}of}(\textbf{city1-1}) \mathrel{\hat{=}} \textbf{city-1} \land \mathsf{city\text{-}of}(\textbf{city1-2}) \mathrel{\hat{=}} \textbf{city-1} \land \ldots$
$\textbf{init}\ [0]\ \forall \mathit{thing}, \mathit{loc}\ [\ \mathsf{at}(\mathit{thing}, \mathit{loc}) \leftrightarrow$
$\quad \mathit{thing}{=}\textbf{package-1} \land \mathit{loc}{=}\textbf{city2-1} \lor$
$\quad \mathit{thing}{=}\textbf{package-2} \land \mathit{loc}{=}\textbf{city1-2} \lor \ldots \lor$
$\quad \mathit{thing}{=}\textbf{truck-1} \land \mathit{loc}{=}\textbf{city1-1} \lor \ldots \lor$
$\quad \mathit{thing}{=}\textbf{plane-1} \land \mathit{loc}{=}\textbf{city4-2}\ ]$
$\textbf{init}\ [0]\ \forall \mathit{obj}, \mathit{vehicle}\ [\ \neg\mathsf{in}(\mathit{obj}, \mathit{vehicle})\ ]$ ■

Example 6.2.8 (Blocks World, continued)
The initial state of the blocks world problem instance from Figure 5.2 on page 119 can be defined as follows:

$\textbf{init}\ [0]\ \mathsf{handempty} \land \forall \mathit{block}\ [\ \neg\mathsf{holding}(\mathit{block})\ ]$
$\textbf{init}\ [0]\ \forall \mathit{block}\ [\ \mathsf{clear}(\mathit{block}) \leftrightarrow \mathit{block}{=}\textbf{C} \lor \mathit{block}{=}\textbf{E}\ ]$
$\textbf{init}\ [0]\ \forall \mathit{block}\ [\ \mathsf{ontable}(\mathit{block}) \leftrightarrow \mathit{block}{=}\textbf{A} \lor \mathit{block}{=}\textbf{D}\ ]$
$\textbf{init}\ [0]\ \forall \mathit{block}_1, \mathit{block}_2\ [\ \mathsf{on}(\mathit{block}_1, \mathit{block}_2) \leftrightarrow$
$\quad \mathit{block}_1{=}\textbf{C} \land \mathit{block}_2{=}\textbf{B} \lor \mathit{block}_1{=}\textbf{B} \land \mathit{block}_2{=}\textbf{A} \lor \mathit{block}_1{=}\textbf{E} \land \mathit{block}_2{=}\textbf{D}\ ]$ ■

6.2.5 What to Achieve: Goals

Classical state-based goals provide a set of constraints that should be satisfied by the final state reached by executing a solution plan, when starting from a given initial state.

In a sense, one could argue that classical goals are subsumed by control formulas and therefore do not have to be supported explicitly in TALplanner. After all, the intention behind control formulas is to provide the ability to place arbitrary constraints on the state sequence generated by executing a solution plan. Such arbitrary constraints must clearly be able to model simple state goals. Instead of requiring that the final state satisfy the formula ϕ, one could require that the state sequence satisfy the formula $\exists t.[t, \infty)\ \phi$ – there should be a timepoint t after which the state goal ϕ holds indefinitely.

Nevertheless, we still consider classical goals to be important enough to warrant a separate statement class. This also enables control rules to refer to the classical goals, providing additional pruning power (next section).

TALplanner supports classical state-based goals, which provide a set of constraints on the state that should be reached by executing a solution plan when starting from a given initial state. Since TAL-C has no statement type for goals, a new statement type has been added to $\mathcal{L}(\mathrm{ND})^*$. This statement type allows goal statements to be specified using a set of fluent formulas describing the set of goal states.

Definition 6.2.7 (Goal Statement)
A *goal statement* in $\mathcal{L}(\mathrm{ND})^*$, labeled goal, is formed from elementary fluent formulas having the form $f \hat{=} \omega$ using conjunction and universal quantification over values, where f is a fluent term containing no nested fluent terms and ω is a value term containing no fluent terms. As a shorthand notation, $f \hat{=}$ **true** can also be written f. ■

The restriction to conjunctive goals with no nested fluent terms is mainly intended to streamline the presentation of the $\mathcal{L}(\mathrm{FL})$ semantics for goal statements and goal expressions (next section). Arbitrary disjunctive and existential goals are in fact allowed in the implementation. A formal semantics for such goals, partly based on formula rewriting during translation into $\mathcal{L}(\mathrm{FL})$, has been presented in earlier publications (Kvarnström & Doherty, 2000b). A cleaner and more elegant (but still equivalent) semantics is expected to be facilitated by future extensions to TAL and $\mathcal{L}(\mathrm{FL})$.

It should be noted that although explicit negations are not allowed in goal statements, negative goals of the form $\neg\mathsf{at}(\mathit{object}, \mathit{location})$ can be written as $\mathsf{at}(\mathit{object}, \mathit{location}) \hat{=}$ **false**. Also note again that it will be possible to express more complex goals, not limited to the final state, using control rules.

For a planner such as TALplanner, where operators can be executed over an extended period of time, there is a question of whether it is sufficient that the planner *visits* a goal state at some point during the execution of an action or whether the planner must achieve a *persisting* goal state at the end of the execution of the final action in the plan. TALplanner currently requires a persisting goal state, though this is not a fundamental property of the planner but a design decision that could easily be changed. This leads to the following translation of goal statements:

- $\mathit{Trans}(\textbf{goal}\ \phi) = \mathit{Trans}(\exists t.[t, \infty)\ \phi)$

After applying an action, the planner tests whether the current narrative entails the conjunction of all goal formulas. If it does, a persisting goal state has been achieved (because ϕ must hold from some arbitrary timepoint t until infinity), and the current plan candidate can be returned as a solution.

Example 6.2.9 (Logistics, Gripper and Blocks, continued)
The goal specification for the logistics problem instance in Figure 5.3 on page 120 can be written as follows:

goal at(**package-1**, **city2-1**) $\wedge$ at(**package-2**, **city1-2**)
goal at(**package-3**, **city1-1**) $\wedge$ at(**package-4**, **city1-1**)
goal at(**package-5**, **city4-2**) $\wedge$ at(**package-6**, **city3-1**)

The following goal specification for a gripper problem instance requires all balls to be in room B and requires all grippers to be free, but does not constrain the location of the robot. This is consistent with the goal for the problem instance in Figure 5.1 on page 118.

goal $\forall ball$ [loc($ball$) $\hat{=}$ **roomB**] $\wedge$ $\forall gripper$ [free($gripper$)]

The following goal statements are one possible description of the blocks world goal state described in Figure 5.2 on page 119.

goal clear(**B**) $\wedge$ on(**B**,**E**) $\wedge$ on(**E**,**C**) $\wedge$ ontable(**C**)
goal clear(**A**) $\wedge$ on(**A**,**D**) $\wedge$ ontable(**D**) ■

6.2.6 The Goal Modality: Querying the Goal

TLPlan control rules use a goal modality to test whether or not the goal of the current problem instance requires a certain formula to hold: If γ is the conjunction of all goal formulas, then the formula goal(ϕ) holds iff $\gamma \models \phi$, where ϕ may contain variables bound outside the goal modality. This is key to the pruning power of control rules, and a formula class similar to the goal modality must therefore be provided by TALplanner.

As discussed in the previous section, this presentation is limited to conjunctive goals. For such goals, the information present in the goal specification can be captured by a simple syntactic transformation during the translation from $\mathcal{L}(\mathrm{ND})^*$ to $\mathcal{L}(\mathrm{FL})$. The intention behind this transformation is to generate, for each original fluent $f : \mathtt{dom}_1 \times \cdots \times \mathtt{dom}_n \rightarrow \mathtt{dom}$ explicitly defined in a narrative, a function $\mathsf{goal}_f : \mathtt{dom}_1 \times \cdots \times \mathtt{dom}_n \times \mathtt{dom} \rightarrow \{\mathbf{true}, \mathbf{false}\}$ that can be queried to determine whether or not the goal forces the fluent to take on a particular value in `dom`. For example, the fluent loc(`obj`) : `location` should give rise to a boolean function $\mathsf{goal}_{\mathsf{loc}}$(`obj`, `location`), where $\mathsf{goal}_{\mathsf{loc}}$(**robby**, **roomA**) holds if and only if the goal entails that **robby** is in **roomA**. This function will then replace the use of a modality in TLPlan.

Because the goal modality is only concerned with the classical state-based goal, which is constant and does not vary over time, a timeless goal_f function would suffice for our intentions. However, TAL does not permit the specification of timeless functions. Therefore, each goal_f function is modeled as a fluent, called a *goal fluent*. We arbitrarily choose to use the value of this fluent at time 0 to represent the constraints placed on the corresponding original fluent by the goal.

The intention is that $[0]\ \mathsf{goal}_f(\overline{x},\omega)$ should be true if and only if the goal γ entails that $f(\overline{x}) \hat{=} \omega$. This is achieved using TAL durational fluents, a somewhat strange name for fluents whose instances automatically revert to a default value when not explicitly forced to take on another value. Each goal fluent is durational with default value **false**. What remains is to force an appropriate subset of all goal fluents to be true, which can be achieved by generating for each goal statement a new TAL dependency constraint where each goal expression $f(\overline{x}) \hat{=} \omega$ is replaced with $I([0]\ \mathsf{goal}_f(\overline{x},\omega) \hat{=} \textbf{true})$.

Goal expressions can now be introduced as a new type of atomic formula in $\mathcal{L}(\mathrm{ND})^*$.

Definition 6.2.8 (Goal Expression)
A *goal expression* is an atomic formula having the form $\mathsf{goal}(f \hat{=} \omega)$, where f is a fluent term and ω is a value term, neither of which contains any occurrences of the *value* function. ■

Goal expressions are translated as follows:

- $\mathit{Trans}(\mathsf{goal}(f(\overline{x}) \hat{=} \omega)) = \mathit{Trans}([0]\ \mathsf{goal}_f(\overline{x},\omega) \hat{=} \textbf{true})$

This notation can easily be extended to allow conjunctions, disjunctions, and quantification within the scope of the goal macro by observing the following equivalences, which hold for conjunctive goals:

- $\mathsf{goal}(\phi \wedge \psi) \equiv \mathsf{goal}(\phi) \wedge \mathsf{goal}(\psi)$
- $\mathsf{goal}(\phi \vee \psi) \equiv \mathsf{goal}(\phi) \vee \mathsf{goal}(\psi)$
- $\mathsf{goal}(\exists x.\phi) \equiv \exists x.\mathsf{goal}(\phi)$
- $\mathsf{goal}(\forall x.\phi) \equiv \forall x.\mathsf{goal}(\phi)$

6.2.7 Doing Something: Operators and Actions

Since TAL-C is a logic for reasoning about action and change, it has a notion of actions that can be used for modeling planning operators. Effects can be specified using the R or I assignment operators, and conditional effects are easily modeled as well.

The only downside of using TAL operator definitions is their flexibility, which gives them a relative lack of structure. For example, there is no distinction between operator preconditions and conditions used for conditional effects, and the fact that there can be many levels of quantification and many levels of effect conditions means that operator analysis is unnecessarily complex. For simplicity, we therefore introduce a more structured operator definition macro, which can be more easily handled by the planner but which can still be translated into a plain $\mathcal{L}(\mathrm{FL})$ action definition. This is in line with the standard TAL practice of preserving the logical base language $\mathcal{L}(\mathrm{FL})$ and its semantics but providing different variations of the high-level macro language $\mathcal{L}(\mathrm{ND})$ that are adapted to special tasks.

Before presenting a formal specification, we will provide an overview of the new operator macro using a concrete example from the gripper domain: The move-to operator, which moves the robot and all balls currently carried by the robot to another room. This operator could be formalized in plain $\mathcal{L}(\mathrm{ND})$ as follows:

action move-to(room)
acs3 $[s,t]$ move-to($room$) $\rightsquigarrow$
$[s]$ loc(**robby**) $\hat{\neq}$ $room \rightarrow$
$R([t]$ loc(**robby**) $\hat{=}$ $room) \wedge$
$\forall ball\ [\ [s]\ \exists gripper\ [$ is-carried-in($ball,gripper$)$] \rightarrow R([t]$ loc($ball$) $\hat{=}$ $room)] \wedge$
$t{=}s{+}1$

Note the constraint $t{=}s{+}1$ which ensures that the action has unit duration. Using the new operator macro in $\mathcal{L}(\mathrm{ND})^*$, the same operator could instead be defined as follows:

operator move-to($room$) :at s
:duration 1
:precond $[s]$ loc(**robby**) $\hat{\neq}$ $room$
:context
:effects $[s{+}1]$ loc(**robby**) := $room$
:context
:forall $ball$
:condition $[s]\ \exists gripper\ [$ is-carried-in($ball$, $gripper$) $]$
:effects $[s{+}1]$ loc($ball$) := $room$

This *operator specification* encapsulates information about a specific operator type. It specifies the name and the formal arguments of the operator, including a temporal variable which serves as a formal invocation timepoint. It also contains a duration specification, required for actions with extended duration, and a precondition which must only refer to the state where the operator is invoked. Finally, there must be a complete specification of all (possibly quantified and/or conditional) effects of the operator. This is provided in terms of one or more *context specifications*, each of which contains one or more *effect specifications*.

Definition 6.2.9 (Operator Specification; Invocation Timepoint Function)
An *operator specification*[2] in $\mathcal{L}(\mathrm{ND})^*$ is a labeled statement having the following form, where o is an operator name, v_1 through v_n are distinct value variables serving as formal parameters, the invocation timepoint s is a temporal variable, the

[2]Redefined in Definition 7.2.1 on page 197 for concurrency.

duration specification τ is a temporal term, the precondition ϕ is a single timepoint formula for s, and c_1 through c_m are context specifications for the invocation timepoint s (as defined below).

- **operator** o$(v_1, \ldots, v_n)$:at s :duration τ :precond ϕ :context c_1 … :context c_m

For brevity, the formal invocation time variable for o as specified by the :at clause will sometimes be denoted by $\mathsf{inv}(o)$.

Omitting the precondition specification (:precond ϕ) is equivalent to specifying :precond `true`. Omitting the duration specification is equivalent to specifying :duration 1. For actions with only one context specification, the :context keyword can be omitted. ■

Notation: Operators are often denoted by o. In a narrative with n operator definitions, operators may be denoted by o^i where the superscript index identifies the operator type $(0 \leq i < n)$. In a plan containing m action instances, operators may be denoted by o_j where the subscript index indentifies the index of the operator within the plan $(0 \leq j < m)$. Consequently, the operator o^i_j is an operator of type i occurring as the j:th action in a plan. Operator arguments are denoted by ω where arbitrary value terms are allowed and by c in concrete plans where only constant value arguments are allowed. Overlines ($\bar{c}$) indicate implicit sequences of arguments.

An operator specification can contain a number of context specifications, each of which encapsulates a set of conditional quantified effects.

Definition 6.2.10 (Context Specification)
A *context specification*[3] for the invocation timepoint s has the following form, where v_1 through v_n are distinct value variables, ϕ is a single timepoint formula for s, and e_1 through e_m are effect expressions for s (as defined below):

- :forall $v_1, \ldots, v_n$:condition ϕ :effects $e_1, \ldots, e_m$

If quantification is not required, the quantifier section (:forall $v_1, \ldots, v_n$) can be omitted. If no context condition is required, the condition section (:condition ϕ) can be omitted; this is equivalent to specifying :condition `true`. ■

An effect expression specifies a single effect taking place at a single timepoint or during an interval. Interval effects are mainly useful in concurrent domains, where more than one operator instance may affect a fluent at any given time, and are interpreted as forcing a fluent to take on a certain value at all timepoints during the specified interval. If the temporal interval is empty or negative ($\tau' < \tau$ below), the fluent will not be affected at all. This is useful in certain special cases; see Section 7.2.3 on page 200 for an example.

[3]Redefined in Definition 7.3.4 on page 202 for resources.

Definition 6.2.11 (Effect Expression)
An *effect expression* for the invocation timepoint s has one of the following forms, where τ and τ' are temporal terms, f is a single timepoint fluent term for s, and ω is a single timepoint value term for s:

- $[s + \tau]\ f := \omega$
- $[s + \tau, s + \tau']\ f := \omega$ ■

Correctness of Operator Specifications

In addition to the syntactical structure specified above, operator definitions must satisfy a small number of additional temporal constraints. Operator durations must be strictly positive, and operator specifications where action effects may take place in or before the invocation state, or after the specified action duration, are not valid. More formally, given an operator with duration δ, and an effect $[s + \tau]\ f := \omega$ or $[s + \tau, s + \tau']\ f := \omega$, it must be the case that $\delta > 0$, $\tau > 0$, $\tau \leq \delta$ and $\tau' \leq \delta$.

Verifying this aspect of operator specifications in advance can be complicated, since both action durations and effect timepoints may depend on the invocation state and cannot be evaluated without access to the specific goal narrative to which an action instance is added. The current implementation verifies correctness for each action added to a plan, ensuring that no invalid plans are generated even if invalid operator specifications are provided as input to the planner.

Action Applicability

It must be possible to determine whether or not an operator instance is applicable in a given goal narrative, during a given temporal interval, with a given set of parameters. In order for a specific instance of an operator to be applicable, the corresponding precondition must be satisfied, the specified duration must correspond to the duration of the interval where the action should be executed, and those conditional or unconditional effects that actually take place should not contradict either each other or effects from other actions already present in the narrative.

Definition 6.2.12 (Applicable Action)
Let $o(\overline{x})$ be an operator with formal arguments $\overline{x}$, invocation timepoint s, precondition ϕ and duration δ. Then, the action $o(\overline{c})$ is *applicable* over the temporal interval $[\tau, \tau']$ in a goal narrative $\mathcal{N}$ iff the following conditions hold:

- $\mathit{Trans}^{+}(\mathcal{N}) \models \mathit{Trans}(\phi[s \mapsto \tau, \overline{x} \mapsto \overline{c}])$
- $\mathit{Trans}^{+}(\mathcal{N}) \models \mathit{Trans}(\tau' - \tau = \delta[s \mapsto \tau, \overline{x} \mapsto \overline{c}])$
- $\mathit{Trans}^{+}(\mathcal{N} \cup \{[\tau, \tau']\ o(\overline{c})\}) \not\models \texttt{false}$ ■

The precondition and the duration constraints are easily checked by evaluating them in the invocation state. Conditional and unconditional effects can simply be applied by adding the new operator instance (action occurrence) to the existing goal narrative – if they are contradictory, the resulting narrative will be inconsistent. Note that because the planner will need to construct an explicit state-based model of the goal narrative, consistency checking is essentially free.

Note the difference in intention between the applicability criteria specified in this section and the correctness criteria in the previous section: An action that does not satisfy applicability criteria is still a valid action, though it cannot be applied in the given context. An action that does not satisfy correctness criteria is always invalid, even if this only occurs in specific contexts.

Context-Dependent Durations in TAL

It should be possible for the duration of each operator to be dependent on the specific arguments with which it is invoked as well as on the current state of the world. For example, the time required to drive between two locations might depend on the distance between these locations. This could be modeled using an integer-valued fluent $\mathsf{distance}(\mathtt{location}, \mathtt{location})$ whose value is interpreted as a duration, except that in the current version of TAL-C the $\mathsf{distance}$ fluent cannot be used where a temporal term is expected, because the value sorts $\mathcal{V}_i$ are considered to be distinct from the temporal sort $\mathcal{T}$.

This is a minor technical issue which can be fixed by a simple change to the current order-sorted type structure of TAL-C, for example by allowing fluents to take values from a finite subsort of the temporal sort. For now, operators with context-dependent duration are supported in TALplanner through a conversion function $\mathsf{maketime}()$ defined through semantic attachment, converting values in a numeric value sort $\mathcal{V}_i$ to timepoints in the temporal sort $\mathcal{T}$. This function can also be used in effect specifications to specify the timepoints at which effects take place. An example of the use of $\mathsf{maketime}()$ is shown in the concurrent logistics domain in Section 7.1.1.

Comparison with PDDL

In recent versions of the commonly used domain definition language PDDL (Fox & Long, 2003; Edelkamp & Hoffmann, 2004), operator durations can be modeled but effects can only take place at the start or the end of a durative action. Like Smith (2003) we argue that allowing effects to take place at arbitrary timepoints permits many processes to be modeled as single operators and that this is in fact a natural way of modeling certain types of processes that cannot be stopped once initiated. The alternative workaround, using multiple operator types with artificial preconditions and effects that ensure the entire chain of required operators is executed in proper order, leads to unnecessary complexity and obscures the true planning domain. TALplanner effect specifications therefore contain a timepoint or an interval of time where an effect should take place, specified as an offset from the operator invocation timepoint.

Another conceptual difference between the TAL semantics and the PDDL semantics relates to the interpretation of timing specifications for action effects. The authors of PDDL2.1 argue that the instantaneous effects present in PDDL2.1 action definitions are in fact only abstractions of a true model where effects take place over intervals of time. Therefore, if an action causes a logical condition ϕ to become true at time τ, this condition is considered a "moving target" and one cannot immediately apply another action that depends on ϕ at the same time τ – though one can apply it at $\tau + \epsilon$ for an arbitrary $\epsilon > 0$.

Though we do recognize the difficulties involved in providing a crisp and well-defined logical model of a real world domain where exact action timings can never be determined in advance, we believe the PDDL2.1 semantics only adresses a very narrow philosophical aspect of this problem, adding unnecessary complexity for no real gain. Because fluent values are only considered to be undetermined at a single instant of time, an effect modifying a fluent f at (for example) time 4 still allows the previous value of f to be used at time 3.999999 and allows the newly assigned value to be used at time 4.000001. Therefore, the fact that the value of f is undetermined at time 4 is *only* relevant for our model of the real world if the true effect did indeed materialize exactly at time 4 – and the difficulty or impossibility of determining this timepoint in advance was the only reason for introducing the "moving target" concept in the first place. If the effect instead materialized at time 4.02, the PDDL semantics allowed another action to rely on the new value too early. If it materialized at time 3.78, the PDDL semantics introduced an unnecessary delay.

A more complete treatment of this issue would involve allowing actions to have incompletely specified durations, where driving between two locations might take between 3.12 and 4.27 units of time, and allowing actions to specify *intervals* of time where fluents are partly or completely undefined up to a specific point in time where an effect is guaranteed to have materialized. The use of intervals rather than instants where fluents are undefined leads to a true increase in expressive power, and is directly supported by TAL though not yet implemented in TALplanner.

Translation into $\mathcal{L}(\mathbf{FL})$

The definition of the *Trans* translation function can be extended as follows in order to translate operator specifications (labeled operator), context specifications, and effect specifications into TAL-C action schemas (labeled acs). Note that $\mathit{Trans}_{\mathsf{con}}$ and $\mathit{Trans}_{\mathsf{eff}}$ generate $\mathcal{L}(\mathrm{ND})$ formulas that are eventually translated into $\mathcal{L}(\mathrm{FL})$ by $\mathit{Trans}_{\mathsf{op}}$.

- $\mathit{Trans}(\text{operator } \mathsf{o}(v_1, \ldots, v_n) \text{ :at } s \text{ :duration } \delta \text{ :precond } \phi$
  $\quad\quad \text{:context } c_1 \ldots \text{:context } c_m) =$
  $\forall s, s', v_1, \ldots, v_n.$
  $\mathit{Occurs}(s, s', \mathsf{o}(v_1, \ldots, v_n)) \rightarrow \mathit{Trans}(\phi \rightarrow \bigwedge_{i=1}^{m} \mathit{Trans}_{\mathsf{con}}(s, c_i))$
- $\mathit{Trans}_{\mathsf{con}}(s, \text{:forall } v_1, \ldots, v_n \text{ :condition } \phi \text{ :effects } \psi_1, \ldots, \psi_m) =$
  $\forall v_1, \ldots, v_n[\phi \rightarrow \bigwedge_{i=1}^{m} \mathit{Trans}_{\mathsf{eff}}(s, \psi_i)]$
- $\mathit{Trans}_{\mathsf{eff}}(s, [s+\tau, s+\tau']\ f := \omega) = I([s+\tau, s+\tau']\ f \mathrel{\hat{=}} \omega)$
- $\mathit{Trans}_{\mathsf{eff}}(s, [s+\tau]\ f := \omega) = I([s+\tau]\ f \mathrel{\hat{=}} \omega)$

Example Operator Definitions

The following examples show operator definitions from the logistics domain and the gripper domain. Further examples, making use of variable action durations as well as other features of TALplanner operator descriptions, can be found in Chapter 7 and in Chapter 9.

Example 6.2.10 (Logistics Domain, continued)
In the logistics domain, packages can be loaded into trucks or airplanes and can naturally also be unloaded. Trucks can drive between arbitrary locations within a single city, and planes can fly between airports. This is usually modeled using a total of six operators, with different operators for different types of vehicles. For example, there are usually two operators for loading packages into vehicles:

operator load-truck(*obj*, *truck*, *loc*) :at *s*
:precond [*s*] at(*obj*, *loc*) ∧ at(*truck*, *loc*)
:effects [*s*+1] at(*obj*, *loc*) := **false**, [*s*+1] in(*obj*, *truck*) := **true**

operator load-plane(*obj*, *plane*, *loc*) :at *s*
:precond [*s*] at(*obj*, *loc*) ∧ at(*plane*, *loc*)
:effects [*s*+1] at(*obj*, *loc*) := **false**, [*s*+1] in(*obj*, *plane*) := **true**

Given a sufficiently flexible type system this is not necessary. TALplanner uses the order-sorted type system of TAL, and both `truck` and `airplane` have been modeled as subtypes of `vehicle`, allowing a single load operator to be used:

operator load(*obj*, *vehicle*, *loc*) :at *s*
:precond [*s*] at(*obj*, *loc*) ∧ at(*vehicle*, *loc*)
:effects [*s*+1] at(*obj*, *loc*) := **false**, [*s*+1] in(*obj*, *vehicle*) := **true**

Similarly, a single unload operator can be used for both types of vehicles. The tasks of driving and flying are sufficiently different to merit separate operator definitions, though, because a truck can drive between arbitrary locations but only within a single city, while an airplane can move between different cities but can only visit locations that are airports.

```
operator   unload($obj, vehicle, loc$) :at $s$
  :precond  [$s$] in($obj, vehicle$) $\wedge$ at($vehicle, loc$)
  :effects  [$s$+1] in($obj, vehicle$) := **false**, [$s$+1] at($obj, loc$) := **true**

operator   drive($truck, loc_1, loc_2$) :at $s$
  :precond  [$s$] at($truck, loc_1$) $\wedge$ city-of($loc_1$) $\hat{=}$ city-of($loc_2$) $\wedge$ $loc_1 \neq loc_2$
  :effects  [$s$+1] at($truck, loc_1$) := **false**, [$s$+1] at($truck, loc_2$) := **true**

operator   fly($plane, airport_1, airport_2$) :at $s$
  :precond  [$s$] at($plane, airport_1$) $\wedge$ $airport_1 \neq airport_2$
  :effects  [$s$+1] at($plane, airport_1$) := **false**, [$s$+1] at($plane, airport_2$) := **true**
```

Note that we use the traditional definitions of the in and at predicates, where objects that are currently inside a vehicle are not considered to be at a location. ■

Example 6.2.11 (Gripper Domain, continued)
Three operators are available to the robot in the gripper domain. The robot can pick up a ball in any free gripper, as long as the ball is in the same location and the robot is not already carrying it. It can always drop a ball that it is carrying. Finally, it can move to another room, which changes not only the location of the robot but also the location of any ball that it is currently carrying, demonstrating the use of quantified conditional effects.. As in the logistics domain, there is no map modeling connections between rooms. Instead, the robot is able to move directly from any room to any other room.

```
operator   pick($ball, gripper$) :at $s$
  :precond  [$s$] loc($ball$) $\hat{=}$ loc(**robby**) $\wedge$ free($gripper$) $\wedge$
               $\neg\exists gripper'$ [ is-carried-in($ball, gripper'$) ]
  :effects  [$s$+1] is-carried-in($ball, gripper$) := **true**,
            [$s$+1] free($gripper$) := **false**

operator   drop($ball, gripper$) :at $s$
  :precond  [$s$] is-carried-in($ball, gripper$)
  :effects  [$s$+1] is-carried-in($ball, gripper$) := **false**,
            [$s$+1] free($gripper$) := **true**

operator   move-to($room$) :at $s$
  :precond  [$s$] loc(**robby**) $\hat{\neq}$ $room$
:context
  :effects [$s$+1] loc(**robby**) := $room$
:context :forall $ball$ :condition [$s$] $\exists gripper$ [ is-carried-in($ball, gripper$) ]
  :effects [$s$+1] loc($ball$) := $room$
```
■

6.2.8 Combining Actions into Plans

A *plan* is an executable set or sequence of actions, and a plan which also entails the goal and all control rules is called a *solution* or a *solution plan*.

Due to the use of actions with non-unit duration in TALplanner, even purely sequential plans must contain timing information. This information will consist of an exact numeric execution interval for each action in the plan, and therefore it seems reasonable to reuse the standard TAL concept of *action occurrences* for this purpose rather than introducing new planning-specific structures.

An action occurrence has the form $[\tau, \tau'] \; o(\overline{\omega})$, denoting the invocation of the operator o with the arguments $\overline{\omega}$ between times τ and τ'. TAL actions must occur over non-empty intervals of time, implying that $\tau < \tau'$. A concrete example for the logistics domain would be $[0, 1]$ load(**package-5**, **truck-1**, **city1-1**).

Since action occurrences include explicit timing information, there is strictly speaking no need to provide additional structure in order to maintain proper ordering and timing relations: Plans can be viewed simply as sets of action occurrences, without loss of information. Despite this, it is sometimes useful to represent sequential plans as sequences or tuples rather than as sets, given that the TALplanner search procedure always adds one action at a time. These two representations will be used interchangeably.

Definition 6.2.13 (Sequential Plan)
A *sequential plan* for a goal narrative $\mathcal{N}$ is a tuple of ground fluent-free action occurrences with the following constraints. First, the empty tuple is a sequential plan for $\mathcal{N}$. Second, given a sequential plan $p = \langle [\tau_1, \tau_1'] \; o_1(\overline{c}_1), \ldots, [\tau_n, \tau_n'] \; o_n(\overline{c}_n) \rangle$ for $\mathcal{N}$, its successors are exactly those sequences adding one new action occurrence $[\tau_{n+1}, \tau_{n+1}'] \; o_{n+1}(\overline{c}_{n+1})$ satisfying the following constraints:

1. Let $\mathcal{N}' = \mathcal{N} \cup \{[\tau_1, \tau_1'] \; o_1(\overline{c}_1), \ldots, [\tau_n, \tau_n'] \; o_n(\overline{c}_n)\}$ be the original goal narrative $\mathcal{N}$ combined with the existing plan. Then, the new action $o_{n+1}(\overline{c}_{n+1})$ must be applicable over the interval $[\tau_{n+1}, \tau_{n+1}']$ in $\mathcal{N}'$. This implies that its preconditions are satisfied, that its effects are not internally inconsistent and do not contradict the effects of the operator instances already present in the sequence, and that the duration $\tau_{n+1}' - \tau_{n+1}$ is consistent with the duration given in the operator specification.

2. The first action starts at time 0: $\tau_1 = 0$.

3. Each action occurrence follows the preceding action occurrence with no gaps: For all $1 < k \leq n$, $\tau_k = \tau_{k-1}'$. ■

Definition 6.2.14 (Sequential Solution)
A *sequential solution*[4] for a goal narrative $\mathcal{N}$ is a sequential plan p for $\mathcal{N}$ such that $\mathit{Trans}^+(\mathcal{N} \cup p) \models \mathit{Trans}(\mathcal{N}_{\text{goal}})$, where Trans^+ is the narrative translation procedure for $\mathcal{L}(\text{ND})^*$ as defined in Section 6.2.10. ■

[4]Redefined in Definition 6.4.2 on page 168 to add support for control rules.

6.2.9 Domains and Problem Instances

A TAL narrative has traditionally been specified as a single structure containing what the planning community would view as two separate structures: A domain definition and a problem instance definition. The implementation of TALplanner has an extended parser which allows the domain definition and the problem instance definition to be specified separately. Though this has some effect on the efficiency of the planner, allowing it to do more extensive pre-processing on the domain definition before being given a sequence of problem instances to solve, this should mainly be considered an implementation detail. For the purposes of this thesis, the information specified to the planner will usually be considered to consist of a single goal narrative.

6.2.10 The Extended Language $\mathcal{L}(\mathbf{ND})^*$

Most of the planning-related language extensions to TAL-C have now been introduced, including new statement classes for operator definitions, goal statements, and definitions for defined fluents. Two additional statement classes remain to be presented: TAL-based control rules with explicit time, labeled control, and control rules using tense operators (introduced as macros in $\mathcal{L}(\text{ND})^*$), labeled tcontrol. These statement classes will be examined in the following sections, together with a more extensive discussion of the properties of control rules and how they are used in TALplanner. Before that, it is time for an overview of the new language and a revision of the TAL-C translation procedure and circumscription policy from Section 2.4.2.

The following definition provides an overview of the statement classes used in the planning-specific TAL macro language $\mathcal{L}(\text{ND})^*$.

Definition 6.2.15 (Narrative Components in $\mathcal{L}(\mathbf{ND})^*$)
A narrative in $\mathcal{L}(\text{ND})^*$ consists of the following statement classes.

- Operator specification statements, labeled **operator** (Definition 6.2.9).
- Fluent definition statements, labeled **define** (Definition 6.2.4).
- Initialization statements, labeled **init** (Definition 6.2.6).
- Action occurrence statements, labeled **occ**. As in plain $\mathcal{L}(\text{ND})$, an action occurrence statement is an occurrence formula $[\tau, \tau']\ \Psi$ where τ and τ' are variable-free temporal terms and Ψ is a variable-free action term.
- Goal statements, labeled **goal** (Definition 6.2.7).
- TAL control statements, labeled **control** (Definition 6.4.1 on page 167).
- Tense control statements, labeled **tcontrol** (Definition 6.5.2 on page 180). ■

Note that action occurrence statements must not be present in the initial goal narrative specified as input to the planner. They are added incrementally as a plan is being built. Also note that the definition above does not include statements related to the vocabulary of a planning domain: Value domains, objects, and features.

Translation. We identify two separate aspects of translating $\mathcal{L}(\text{ND})^*$ narratives. The *Trans* translation function is responsible for translating a single unstructured formula into $\mathcal{L}(\text{FL})$, while the narrative translation procedure discussed in Section 2.4.2, denoted by Trans^+, is responsible for translating a complete structured narrative consisting of a number of labeled statements belonging to a predefined set of statement classes. This separation is necessary due to the use of filtered circumscription, where the *Occlude* and *Occurs* predicates should be circumscribed relative to only a subset of the statement classes used in $\mathcal{L}(\text{ND})^*$.

As new macros have been added to the new language $\mathcal{L}(\text{ND})^*$, the *Trans* function has been extended incrementally. The following list summarizes the extensions to the translation function.

- $\mathit{Trans}([t]f(v_1,\ldots,v_n) : \phi) = \mathit{Trans}(\forall t, v_1,\ldots,v_n[[t]f(v_1,\ldots,v_n) \leftrightarrow \phi])$
- $\mathit{Trans}([t]f(v_1,\ldots,v_n) : \omega) = \mathit{Trans}(\forall t, v_1,\ldots,v_n[[t]f(v_1,\ldots,v_n) \hat{=} \omega])$
- $\mathit{Trans}(\textbf{goal}\ \phi) = \mathit{Trans}(\exists t.[t,\infty)\ \phi)$, for goal statements ϕ
- $\mathit{Trans}(\mathsf{goal}(f(\overline{x}) \hat{=} v)) = \mathit{Trans}([0]\ \mathsf{goal}_f(\overline{x}, v) \hat{=} \textbf{true})$
- $\mathit{Trans}(\textbf{operator}\ \mathsf{o}(v_1,\ldots,v_n)\ \mathsf{:at}\ s\ \mathsf{:duration}\ \delta\ \mathsf{:precond}\ \phi$
  $\mathsf{:context}\ c_1 \ldots \mathsf{:context}\ c_m) =$
  $\forall s, s', v_1,\ldots,v_n.$
  $\mathit{Occurs}(s, s', \mathsf{o}(v_1,\ldots,v_n)) \rightarrow \mathit{Trans}(\phi \rightarrow \bigwedge_{i=1}^{m} \mathit{Trans}_{\mathsf{con}}(s, c_i))$.
- $\mathit{Trans}_{\mathsf{con}}(s, \mathsf{:forall}\ v_1,\ldots,v_n\ \mathsf{:condition}\ \phi\ \mathsf{:effects}\ \psi_1,\ldots,\psi_m) =$
  $\forall v_1,\ldots,v_n[\phi \rightarrow \bigwedge_{i=1}^{m} \mathit{Trans}_{\mathsf{eff}}(s, \psi_i)]$
- $\mathit{Trans}_{\mathsf{eff}}(s, [s+\tau, s+\tau']\ f := \omega) = I([s+\tau, s+\tau']\ f \hat{=} \omega)$
- $\mathit{Trans}_{\mathsf{eff}}(s, [s+\tau]\ f := \omega) = I([s+\tau]\ f \hat{=} \omega)$

What remains is to extend the narrative translation procedure for the new statement classes. To do this, we also add three new $\mathcal{L}(\text{FL})$ statement classes for control formulas, tense control formulas and goals, using the same labels as in $\mathcal{L}(\text{ND})^*$. Though this may appear to be in conflict with our desire to keep the base logic $\mathcal{L}(\text{FL})$ unchanged, the addition of new statement classes is in fact a very minor modification that keeps the underlying structure and circumscription policy intact.

In the following,

- $\mathcal{N}$ denotes the collection of narrative statements contained in a goal narrative in $\mathcal{L}(\text{ND})^*$.
- $\mathcal{N}_{\mathsf{per}}$ denotes the set of implicitly specified persistence statements in $\mathcal{N}$ characterizing the behavior of persistent and durational fluents, where defined fluents are assumed to be dynamic (neither persistent nor durational), fluents representing the goal state are durational as specified in Section 6.2.6, and all other fluents are assumed to be persistent. $\Gamma_{\mathsf{per}} = \mathit{Trans}(\mathcal{N}_{\mathsf{per}})$ denotes the corresponding translation into $\mathcal{L}(\text{FL})$.
- $\mathcal{N}_{\mathsf{init}}$ denotes the set of initialization statements in $\mathcal{N}$characterizing the initial state, and $\Gamma_{\mathsf{obs}} = \mathit{Trans}(\mathcal{N}_{\mathsf{init}})$ the corresponding translation into observation statements in $\mathcal{L}(\text{FL})$.
- $\mathcal{N}_{\mathsf{occ}}$ denotes the set of action occurrence statements in $\mathcal{N}$, for a goal narrative where the planning algorithm has already added a set of action occurrences, and $\Gamma_{\mathsf{occ}} = \mathit{Trans}(\mathcal{N}_{\mathsf{occ}})$ the corresponding translation into $\mathcal{L}(\text{FL})$.
- $\mathcal{N}_{\mathsf{op}}$ denotes the set of operator definition statements in $\mathcal{N}$, and $\Gamma_{\mathsf{acs}} = \mathit{Trans}(\mathcal{N}_{\mathsf{op}})$ the corresponding translation into action type specifications in $\mathcal{L}(\text{FL})$.
- $\mathcal{N}_{\mathsf{control}}$ denotes the set of TAL-based control formulas in $\mathcal{N}$, and $\Gamma_{\mathsf{control}} = \mathit{Trans}(\mathcal{N}_{\mathsf{control}})$ the corresponding translation into static formulas in $\mathcal{L}(\text{FL})$.
- $\mathcal{N}_{\mathsf{tcontrol}}$ denotes the set of tense control formulas in $\mathcal{N}$, and $\Gamma_{\mathsf{tcontrol}} = \mathit{Trans}(\mathcal{N}_{\mathsf{tcontrol}})$ the corresponding translation into static formulas in $\mathcal{L}(\text{FL})$.
- $\mathcal{N}_{\mathsf{define}}$ denotes the set of fluent definition statements in $\mathcal{N}$, and $\Gamma_{\mathsf{domc}} = \mathit{Trans}(\mathcal{N}_{\mathsf{define}})$ the corresponding translation into domain constraints in $\mathcal{L}(\text{FL})$.
- $\mathcal{N}_{\mathsf{goal}}$ denotes the set of goal statements in $\mathcal{N}$, and $\Gamma_{\mathsf{goal}} = \mathit{Trans}(\mathcal{N}_{\mathsf{goal}})$ the corresponding translation into static formulas in $\mathcal{L}(\text{FL})$.
- $\mathcal{N}_{\mathsf{depc}}$ denotes the set of dependency constraints providing values for goal fluents as defined in Section 6.2.6, and $\Gamma_{\mathsf{depc}} = \mathit{Trans}(\mathcal{N}_{\mathsf{depc}})$ the corresponding translation into dependency constraints in $\mathcal{L}(\text{FL})$.
- Γ_{fnd} denotes the set of foundational axioms in $\mathcal{L}(\text{FL})$, which contains unique names axioms, unique values axioms, etc.
- Γ_{time} denotes the set of axioms representing the temporal base structure. Since the timepoints in TAL-C use the natural numbers structure, we use the Peano axioms without multiplication.

The *Occlude* predicate is circumscribed relative to the action definitions in Γ_{acs} and the dependency constraints in Γ_{depc} with all other predicates fixed, and *Occurs* is circumscribed relative to the action occurrence formulas in Γ_{occ} with all other predicates fixed. Due to structural constraints on $\mathcal{L}(\mathrm{ND})$ statements, quantifier elimination techniques can then be used to translate the two second-order circumscriptive theories into logically equivalent first-order theories (Doherty et al., 1998; Doherty, 1996), denoted by $Circ(\Gamma_{\mathsf{acs}} \wedge \Gamma_{\mathsf{depc}}; Occlude)$ and $Circ(\Gamma_{\mathsf{occ}}; Occurs)$, respectively.

The two resulting theories are combined and filtered with the $\mathcal{L}(\mathrm{FL})$ translations of the persistence statements in Γ_{per} (forcing persistent and durational fluents to adhere to the persistence or default value assumptions), the domain constraints in Γ_{domc}, and the observations and timing constraints in Γ_{obs}, yielding the theory $\Gamma' = \Gamma_{\mathsf{per}} \wedge \Gamma_{\mathsf{obs}} \wedge \Gamma_{\mathsf{domc}} \wedge Circ(\Gamma_{\mathsf{occ}}; Occurs) \wedge Circ(\Gamma_{\mathsf{depc}} \wedge \Gamma_{\mathsf{acs}}; Occlude)$. Adding the $\mathcal{L}(\mathrm{FL})$ foundational axioms in Γ_{fnd} then yields the theory $\Delta = \Gamma' \wedge \Gamma_{\mathsf{fnd}}$.

The theory Δ is still a first-order theory, but lacks one important component: There is no formal characterization of the linear discrete temporal structure used by TAL. There are two alternatives: One can use an interpreted theory for the temporal structure, or an axiomatization can be added in the shape of a second-order theory Γ_{time} corresponding to the Peano axioms without multiplication.

In the remainder of the thesis, $Trans^{+}(\mathcal{N})$ will denote the result of translating the goal narrative $\mathcal{N}$ into $\mathcal{L}(\mathrm{FL})$ and applying this filtered circumscription policy. The $\mathcal{L}(\mathrm{ND})^{*}$ formula γ is preferentially entailed by the $\mathcal{L}(\mathrm{ND})^{*}$ goal narrative $\mathcal{N}$ iff $Trans^{+}(\mathcal{N}) \models Trans(\gamma)$.

Note that whereas most statement types in TAL provide information about what is definitely the case in a general domain or in a particular problem instance, goals and control formulas are statements about the *desired* state of the world. Therefore, though goals and control formulas are part of a translated narrative, they are not used when determining what is entailed by this narrative – obviously, that would allow the planner to immediately conclude that the goal is satisfied. In other words, these formulas are present in the translation in the shape of $\Gamma_{\mathsf{control}}$, $\Gamma_{\mathsf{tcontrol}}$ and Γ_{goal}, but they are not part of $Trans^{+}(\mathcal{N})$. TALplanner uses the translated formulas during the planning phase, testing whether or not they are entailed by the $Trans^{+}(\mathcal{N} \cup p)$, where p is a set of action occurrences representing a plan.

Notation. We will say that a set of action occurrences p entails a formula ϕ iff $Trans^{+}(\mathcal{N} \cup p) \models Trans(\phi)$, where the narrative $\mathcal{N}$ is often to be understood from the context.

6.3 The Basic TALplanner Algorithm

The TALplanner algorithm is based on the use of a forward-chaining search procedure, where the search tree is usually traversed using depth first search. Although this procedure would be extremely inefficient without the addition of search control knowledge, it is still a complete planning algorithm, due to the use of cycle checking together with the restriction to finite domains where only a finite number of world states are possible.

The definition of depth first search is quite trivial. Nevertheless, we will now present a version of the TALplanner depth first search procedure. This will serve as a concrete basis for several definitions as well as an extensible skeleton to which control rule checking will eventually be added. Explanations follow after this definition.

Definition 6.3.1 (TALplanner Without Control)
Input: A goal narrative $\mathcal{N}$.
Output: A plan narrative which entails the goal $\mathcal{N}_{\mathsf{goal}}$.

```
1  procedure TALplanner-without-control($\mathcal{N}$)
2  $\gamma \leftarrow \bigwedge \mathcal{N}_{\mathsf{goal}}$                         Conjunction of all goal statements
3  node $\leftarrow \langle 0, \langle\rangle\rangle$                              ⟨next invocation time, plan⟩
4  Open $\leftarrow \langle \text{node} \rangle$                                  Stack (depth first search)
5  while Open $\neq \langle\rangle$ do
6     $\langle \tau, p \rangle \leftarrow$ pop(Open)                                Current plan candidate
7     $\mathcal{N}'' \leftarrow \mathcal{N} \cup p$                                Complete goal narrative with plan
8     if $Trans^{+}(\mathcal{N}'') \models \texttt{false}$ then backtrack          Consistency check (def. 6.2.12)
9     if sequential-cycle-check($\mathcal{N}'', \tau$) then backtrack              (defined below)
10    if $Trans^{+}(\mathcal{N}'') \models Trans(\gamma)$ then                     Goal entailed
11       return $\mathcal{N}''$
12    else                                                                  Not a solution, but check children
13       for all actions $\mathbf{A} = [\tau, \tau'] \; o^{i}(\bar{c})$ applicable over $[\tau, \tau']$ in $\mathcal{N}''$ do
14          push $\langle \tau', \langle p; \mathbf{A} \rangle \rangle$ onto Open
15 fail                                                                                                ■
```

The goal narrative provided as input to TALplanner encapsulates both a planning domain definition and a problem instance specification. The TALplanner algorithm begins by extracting the goal statements from this goal narrative and generating an initial search node, which contains the end time of the last action in the plan, which is also the time when the next new action should be invoked given that sequential plans are being generated (0 in the initial node), and the current plan candidate (the empty tuple). The initial search node is pushed onto a stack in order to maintain a depth first search order.

As long as the stack is not empty, a search node $\langle \tau, p \rangle$ is popped from the stack. Adding the current plan candidate p to the original goal narrative $\mathcal{N}$ yields a narrative $\mathcal{N}''$ which may or may not be a solution to the current planning problem.[5]

The planner must then test whether the plan has inconsistent effects, leading to an inconsistent narrative. Note again that since the planner must build an explicit model of the state sequence generated by the current plan, inconsistency checking is essentially free. TALplanner also tests whether the current plan has a state cycle (see also the detailed description under "cycle checking" below). If the plan is

[5]An intermediate narrative called $\mathcal{N}'$ will be added in the next version of the algorithm (Definition 6.4.4 on page 170).

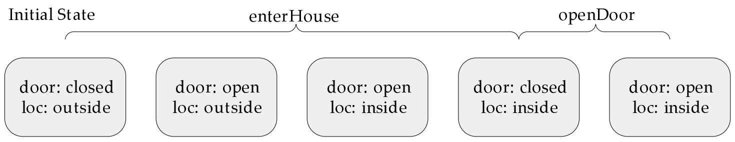


Figure 6.2: Cycle Checking: Not Inside Actions

inconsistent or leads to a state cycle, it must be discarded, and the planner backtracks, retrieving another search node from the stack. Otherwise, the plan is either a solution or a valid inner node whose children should be examined. The plan is a solution if and only if the complete goal narrative $\mathcal{N}''$ (translated into $\mathcal{L}(\mathrm{FL})$, augmented with foundational axioms and circumscribed according to the TAL circumscription policy) entails the goal (translated into $\mathcal{L}(\mathrm{FL})$). If this is not the case, the planner should generate new search nodes for those actions that are applicable at the next action invocation timepoint τ, pushing the new nodes onto the depth first search stack.

Assuming the planner succeeds in finding a plan for $\mathcal{N}$, the result is the $\mathcal{L}(\mathrm{ND})^*$ narrative $\mathcal{N}'' = \mathcal{N} \cup p$, where p is the set of action occurrences (plan steps) generated by the planning algorithm.

6.3.1 Cycle Checking

TALplanner allows the use of cycle checking to prune the search tree.

Let us first consider cycle checking in the context of sequential planning with single-step operators, without control rules. In this setting, each action appended to a plan gives rise to exactly one new state. If this state has already been visited at an earlier point in the plan, there is a state cycle. It is clear that for any solution that contains this cycle, there is a shorter solution where the cycle – or more correctly, the subsequence of actions that caused the cycle – has been removed. Therefore, any action giving rise to a new state cycle can be pruned, without loss of correctness.

Then consider operators with extended duration. If a new action turns out to produce a final state that was previously *temporarily* achieved at an inner point during the execution of an action, then the argument above is no longer valid. Figure 6.2 shows a domain where there is a temporally extended action for entering a house, where as a first step the door is opened, after which the person steps inside and the door is closed. There is also a separate action for opening the door without stepping over the threshold. Executing these actions in sequence produces a sequence of four new states to be added after the initial state. The third and fifth states in the resulting state sequence are identical, but this does not mean that the plan should be pruned. The state cycle does not correspond to a specific subsequence of actions, but to the tail of one action (after the timepoint when the state was temporarily achieved) followed by a subsequence of complete actions. Clearly it is impossible to remove a partial action, and consequently the existence of a solution containing this cycle does not imply the existence of a shorter solution. Therefore, the cycle checking algorithm below must only consider those states that are generated at the end of an action occurrence.

Definition 6.3.2 (Sequential Cycle Checker)
Input: A goal narrative $\mathcal{N}$ and the end timepoint of the last action t_{max}.
Output: `true` if the plan should be discarded, `false` otherwise.

```
1 procedure sequential-cycle-check($\mathcal{N}, t_{\mathsf{max}}$)
2 for $t$ from 0 to $t_{\mathsf{max}} - 1$ do
3   if $\mathcal{N}$ contains no action occurrence $[\tau, \tau']\ o(\overline{\omega})$ where $\tau < t < \tau'$ then
4     if the state at $t$ is identical to the state at $t_{\mathsf{max}}$ then
5       return true
6 return false
```
■

Cycle checking will be revisited for concurrent plans with resources in Section 7.4.

6.3.2 Implementation Notes

The algorithm in Definition 6.3.1 above is necessarily abstracted from the current implementation of TALplanner. A few important implementation details also deserve a brief mention here.

Formula evaluation in state sequences. From the algorithm description above, it may appear that TALplanner uses theorem proving techniques to test whether a goal narrative entails a formula. This is not the case. Instead, an explicit state sequence is generated, and formula evaluation techniques are used to test whether or not a formula is entailed by the corresponding narrative. The state sequence is generated incrementally as the search space is explored, with extensive structure sharing and reuse in order to minimize memory usage.

First order representation. The fact that TALplanner always maintains a first order representation, as opposed to generating ground instances of all actions and fluents, has proven quite useful in many parts of the planner. For example, when the planner tests which instances of an action are applicable in each state, it separates preconditions into parts dependent on no arguments, the first argument, the first two arguments, and so on. In the blocks world, the precondition of the pickup operator includes the condition handempty. If handempty is false, the planner can immediately conclude that no instance of pickup is applicable. Similarly, the precondition of stack requires that the destination is clear, allowing the planner to iterate over only those destinations that satisfy the clear predicate when searching for applicable instances of stack. Suitable candidates for iteration are chosen automatically, with automatic rearrangement of the order of operator arguments to improve iteration performance in certain cases.

Generating search nodes. The last two lines in the planning algorithm above examine all potentially applicable operator instances, explicitly generating one new search node for each applicable instance. Given sufficiently strong search control knowledge, only a few of these nodes may ever have to be visited, and the time spent testing the applicability of the remaining nodes is wasted. Therefore, the TALplanner implementation instead generates a single higher-level search node from which concrete search nodes can be generated on demand for a single operator type at a time. It would also be possible to take this scheme one step further and examine only a single operator *instance* at a time, but this would negate many of the performance advantages of retaining a first-order operator representation as discussed in the previous paragraph.

6.4 TALplanner with TAL-based Control Rules

Though TALplanner can emulate the tense control formulas and progression algorithm used by TLPlan, most of our research has been focused on the use of pure TAL-based control formulas. The first part of this section presents a semantics for such formulas, specifying when a search node should or should not be pruned.

An important property of formula progression is the fact that it allows control formulas to be checked incrementally as each search node adds one or more new states to the state sequence of the parent node. The second part of this section presents a method for pre-processing pure TAL-based control formulas, generating a new set of constraints that can also be checked incrementally in each search node. The method is effective for several formula classes that are commonly used for domain-dependent control, and for these classes, formula evaluation performance is approximately on par with formula progression performance with the benefit of not requiring the storage of progressed control formulas in each search node, thereby reducing memory requirements. More importantly, the method also provides the basic framework supporting the domain analysis techniques presented in Chapter 8, which can provide dramatic performance improvements.

6.4.1 The Semantics of TAL Control Formulas

In order to provide a clean semantics for TAL-based control rules, each control rule will be viewed as a temporally extended goal that must be satisfied (entailed) by the final plan narrative generated by TALplanner. This is a difference from TLPlan, where control rules are only viewed as goal guidance: In TLPlan, generated solutions do not necessarily entail all control rules, due to the use of a progression algorithm which is not strong enough to detect all types of control rule violations.[6]

[6]For example, TLPlan can easily generate plans given the control rule $\Diamond$ `false`, which cannot possibly be satisfied by any plan.

Definition 6.4.1 (TAL Control Rule, TAL Control Statement)
A *TAL control rule* is a static formula. A *TAL control statement*, labeled **control**, consists of a TAL control rule. ■

The definition of sequential solution (Section 6.2.8) must now be amended to account for the fact that solutions must entail all control rules.

Definition 6.4.2 (Sequential Solution)
A *sequential solution* for a goal narrative $\mathcal{N}$ is a sequential plan p for $\mathcal{N}$ such that $\mathit{Trans}^+(\mathcal{N} \cup p) \models \mathit{Trans}(\mathcal{N}_{\mathsf{goal}} \wedge \mathcal{N}_{\mathsf{control}})$. ■

As can be seen in the examples below, the implementation allows control rules to be named as an aid to determining which control rules cause the planner to backtrack. Names have no semantics and are strictly speaking not part of the $\mathcal{L}(\mathrm{ND})^*$ language.

Example 6.4.1 (Control Rules for the Logistics Domain)
The following three simple control rules for the logistics domain are inspired by the control rules used by TLPlan. An object should only be loaded into a plane if a plane is required to move it, that is, if the goal requires it to be at a location in another city. If an object has been unloaded from a plane, it must be the case that the object should be in the current city. If an object is at its destination, it should not be moved.

control :name "only-load-into-plane-when-necessary"
$\forall t, obj, plane, loc.$
$[t]\ \neg\mathsf{in}(obj, plane) \wedge \mathsf{at}(obj, loc) \wedge [t{+}1]\ \mathsf{in}(obj, plane) \rightarrow$
$\exists loc'\ [\ \mathsf{goal}(\mathsf{at}(obj, loc')) \wedge [t]\ \mathsf{city\text{-}of}(loc) \not\hat{=} \mathsf{city\text{-}of}(loc')\ ]$

control :name "only-unload-from-plane-when-necessary"
$\forall t, obj, plane, loc.$
$[t]\ \mathsf{in}(obj, plane) \wedge \mathsf{at}(plane, loc) \wedge [t{+}1]\ \neg\mathsf{in}(obj, plane) \rightarrow$
$\exists loc'\ [\ \mathsf{goal}(\mathsf{at}(obj, loc')) \wedge [t]\ \mathsf{city\text{-}of}(loc) \hat{=} \mathsf{city\text{-}of}(loc')\ ]$

control :name "objects-remain-at-destinations"
$\forall t, obj, loc.$
$[t]\ \mathsf{at}(obj, loc) \wedge \mathsf{goal}(\mathsf{at}(obj, loc)) \rightarrow [t{+}1]\ \mathsf{at}(obj, loc)$ ■

Further control rule examples will be shown in Chapter 9.

6.4.2 Using Control Rules for Pruning

Any given node in a search tree, or any given plan, can be viewed from two quite different perspectives.

Nodes as representatives for subtrees. In some cases, we are interested in what is true in the current plan candidate and all potential descendants. This is the case when testing whether or not a plan should be pruned from the search tree. Given a search node n corresponding to a plan p that satisfies all control formulas, it is not necessarily the case that all ancestors of n (all prefixes of p) satisfy these formulas. Conversely, a search node that violates a control formula may have descendants that satisfy all control formulas. In order to preserve all valid solutions in the search tree, the planner should therefore only prune a node if it can prove that the node *and all its descendants* must violate a control rule, taking all possible extensions of the current plan into consideration. This leads to the question of which facts that are true in the current plan candidate will remain true in all descendants.

Consider the case when a search node $\langle \tau, p \rangle$ has just been retrieved from the search stack. The last action in p is executed during the interval $[\tau_0, \tau]$ for some τ_0. All children of this node will contain one new action invoked at time τ (the last two lines in the planning algorithm). Due to the definition of TALplanner operators, the effects of this action must take place strictly later than τ. The children of this action, in turn, must be invoked at timepoints strictly later than τ and therefore also cannot have effects at or before τ. Thus, the state sequence at $[0, \tau]$ is "fixed", in the sense that it must remain the same for the plan p and all descendants. If the planner can prove that a control rule is violated using only facts about fluents in the interval $[0, \tau]$, then the control rule will definitely remain violated in all descendant search nodes.

The fact that fluent values are only known in the interval $[0, \tau]$ can also be described in terms of lack of any knowledge about the "future" timepoints in (τ, ∞). In TLPlan, this lack of knowledge is handled implicitly using a formula progression algorithm, which step by step evaluates control formulas through those states that become fixed, stopping before the unfixed future is reached. TALplanner instead uses an explicit declarative model of its incomplete knowledge, which can be succinctly captured by occluding all fluents in the narrative after time τ, thereby releasing them from the inertia assumption. This is done in the definition of $\mathcal{N}'$ below, using the occlude-all-after function. Because the planner no longer makes the inertia assumption after τ, the only facts entailed by the narrative are those facts that are explicitly specified in the narrative and those facts that are implied by applying inertia in $[0, \tau]$.

Explicitly modeling incomplete knowledge will be particularly useful in the concurrent version of TALplanner. If the latest action occurrence in a plan takes place at $[\tau_0, \tau]$, the sequential planner can immediately assume complete knowledge of fluent values at all timepoints up to τ, because no new actions can be added earlier than τ. The concurrent planner, on the other hand, can still add new actions as early as τ_0. It will still have complete knowledge up to τ_0, but only *partial* knowledge about later states. The fact that this partial knowledge is correctly modeled and can be taken advantage of is essential to the efficiency of the concurrent planner.

Definition 6.4.3 (occlude-all-after)
By occlude-all-after$(\mathcal{N}, \tau)$ we will denote the finite collection of narrative statements $\{\textbf{dep } X((\tau, \infty)\ f) \mid f \text{ is a fluent in } \mathcal{N}\}$ occluding all fluents in the narrative $\mathcal{N}$. ■

Nodes as plan candidates. In some cases, we are interested in what is true in the current plan candidate, under the assumption that no further changes will ever be made. This is the case when testing whether or not a plan entails the goal of the current planning problem instance: If it does, then it will be returned without further modification. The information available in this plan is then equivalent to that available in the original goal narrative $\mathcal{N}$ together with the current plan p, translated into $\mathcal{L}(\mathrm{FL})$ using the *Trans*$^+$ translation procedure which also performs circumscription and adds the required TAL foundational axioms. One vital property of this translation is that the inertia (persistence) assumption can be applied at all timepoints, allowing the planner to infer a specific value for any fluent at any arbitrary point in time. This variation of the goal narrative is denoted by $\mathcal{N}''$ below.

Given this background, we can now extend the TALplanner algorithm by introducing control knowledge. Lines that are modified from the previous version are marked with a hollow triangle (▷). New lines are marked with a filled triangle (▶).

Definition 6.4.4 (TALplanner With Naive Control)
Input: A goal narrative $\mathcal{N}$.
Output: A plan narrative entailing the goal $\mathcal{N}_{\mathsf{goal}}$ and the control formulas $\mathcal{N}_{\mathsf{control}}$.

```
  1  procedure TALplanner-naive-control($\mathcal{N}$)
  2  $\gamma \leftarrow \bigwedge \mathcal{N}_{\mathsf{goal}}$                         Conjunction of all goal statements
 ▶3  $\phi \leftarrow \bigwedge \mathcal{N}_{\mathsf{control}}$                      Conjunction of all TAL control rules
  4  node $\leftarrow \langle 0, \langle\rangle \rangle$                          ⟨next invocation time, plan⟩
  5  Open $\leftarrow \langle \mathsf{node} \rangle$                          Stack (depth first search)
  6  while Open $\neq \langle\rangle$ do
  7    $\langle \tau, p \rangle \leftarrow$ pop(Open)                             Current plan candidate
 ▶8    $\mathcal{N}' \leftarrow \mathcal{N} \cup p \cup$ occlude-all-after($\mathcal{N}, \tau$)       No knowledge about future
 ▶9    if $Trans^+(\mathcal{N}') \models \neg Trans(\phi)$ then backtrack               Control violated
 10    $\mathcal{N}'' \leftarrow \mathcal{N} \cup p$                              Narrative with complete knowledge
 11    if $Trans^+(\mathcal{N}'') \models$ false then backtrack          Consistency check (def. 6.2.12)
 12    if sequential-cycle-check($\mathcal{N}'', \tau$) then backtrack
▷13    if $Trans^+(\mathcal{N}'') \models Trans(\gamma \wedge \phi)$ then              Goal and control entailed
 14      return $\mathcal{N}''$
 15    else                                      Not a solution, but check children
 16      for all actions $\mathbf{A} = [\tau, \tau'] \, o^i(\bar{c})$ applicable over $[\tau, \tau']$ in $\mathcal{N}'$ do
 17        push $\langle \tau', \langle p; \mathbf{A} \rangle \rangle$ onto Open
 18  fail                                                                   ■
```

Compared to the TALplanner algorithm without control, this algorithm introduces a new narrative version $\mathcal{N}'$, constructed by occluding all fluents after time τ, the end of the last action in the plan. This provides the planner with a correct view of those facts that are true not only in the current plan but also in all descendants. This narrative is used for testing whether there is an inconsistency as well as when checking for state cycles, as in the previous version of the algorithm. It is also used to test whether control rules are violated in this plan and all extensions to the plan, in which case the planner must backtrack. When testing whether a plan should be accepted as a solution, though, complete knowledge about the future should be assumed. As in the previous algorithm, this is done done using the narrative variation $\mathcal{N}''$, which does not add occlusion for all fluents and therefore retains the inertia assumption for all fluents after the end of the last action in the plan.

For intermediate search nodes, the planner only tests whether the negation of a control formula is entailed by a plan. The opposite case, where the control formula itself is entailed by an intermediate search node and could have been removed from the narrative to avoid re-evaluation in descendant nodes, is quite uncommon: Almost all control formulas refer to conditions that must hold throughout the execution of a plan, and such properties are unlikely to be proven to hold when only a prefix of the final solution is being examined. Therefore, the planner only tests whether control formulas ϕ are entailed when it appears that a solution may have been found, that is, after verifying that the goal statements γ are satisfied by the current plan candidate.

6.4.3 Testing Control Formulas Incrementally

Though the naive TALplanner algorithm specified above satisfies the desired semantics for TAL-based control rules, it does so quite inefficiently: It re-evaluates all control formulas in each search node, failing to make use of the fact that adding a new action to a plan gives rise to one or more new states but leaves a large prefix of the state sequence intact. For better performance, control formulas should instead be evaluated in an incremental manner.

In TLPlan, this is achieved using a progression algorithm. Each time an action is added to a plan, the modal control formula associated with the parent search node is progressed through the new states generated by this action, yielding a new control formula to be used as a label for the newly generated search node. If the progression algorithm returns `false`, the planner can immediately backtrack, knowing that the formula is also violated in all descendant nodes.

This method is suitable for a number of logics that use future tense operators, and is reasonably efficient in that it ensures that fluents do not need to be evaluated in newly generated states more than once. However, it is less suitable for TAL formulas with explicit time, and it also has a secondary weakness in that fluents occurring in a progressed formula need to be evaluated *at least* once in every newly generated state. For TALplanner, we have instead constructed a new framework for incrementally testing TAL-based control formulas, which will eventually lead to new optimization opportunities where it may be shown in advance that a formula cannot be violated by a given operator (Chapter 8). The formulas generated in this framework are called *pruning constraints*.

Definition 6.4.5 (Pruning Constraints)
A tuple of *pruning constraints* $\langle \mathsf{init}, \mathsf{incr}, \mathsf{final} \rangle$ for a goal narrative $\mathcal{N}$ consists of the following:

- One set of *initial pruning constraints* init.
- For each operator type o^i in $\mathcal{N}$ with formal invocation timepoint s and formal arguments $x_i^1, \ldots, x_i^{m_i}$ a set $\mathsf{incr}_i(s, x_i^1, \ldots, x_i^{m_i})$ of *incremental pruning constraints* where the variables indicated in parentheses may occur free in the constraints.
- One set of *final pruning constraints* final. ■

The final pruning constraints should generally be applied relative to the last end timepoint of any action in a solution plan. A new temporal constant t_{max} is defined for this purpose. The value of t_{max} cannot be determined until a final solution is accepted. This constant will therefore only be given a value when a plan is considered as a solution candidate, under the assumption that no further changes will be made. This can be achieved either by adding the corresponding observation statement to $\mathcal{N}''$, as done in the modified TALplanner algorithm in Definition 6.4.8 below, or by substituting t_{max} with its value before evaluating the formulas in final.

Definition 6.4.6 (Temporal Constant t_{max})
Let $p = \langle [\tau_1, \tau_1'] \; o_1(\bar{c}_1), \ldots, [\tau_n, \tau_n'] \; o_n(\bar{c}_n) \rangle$ be a sequential or concurrent solution. Then, the temporal constant t_{max} should denote the maximum of all end timepoints of actions in this plan: $t_{\mathsf{max}} = \max_{i=1}^{n} \tau_i'$. ■

Pruning constraints should completely replace control formulas, and the set of constraints generated from a control formula must therefore have identical pruning power. This concept can be formalized as follows.

Definition 6.4.7 (Valid Pruning Constraints)
The pruning constraints $\langle \mathsf{init}, \mathsf{incr}, \mathsf{final} \rangle$ are *valid* pruning constraints for a goal narrative $\mathcal{N}$ with TAL-based control formulas $\mathcal{N}_{\mathsf{control}}$ iff for every plan candidate $p = \langle [\tau_1, \tau_1'] \; o_1^{i_1}(\bar{c}_1), \ldots, [\tau_n, \tau_n'] \; o_n^{i_n}(\bar{c}_n) \rangle$ for $\mathcal{N}$, the conjunction $\bigwedge \mathcal{N}_{\mathsf{control}}$ is equivalent to $\mathsf{init} \wedge \bigwedge_{k=1}^{n} \mathsf{incr}_{i_k}[\mathsf{inv}(o^{i_k}) \mapsto \tau_k, \bar{x}_{i_k} \mapsto \bar{c}_k] \wedge \mathsf{final}[t_{\mathsf{max}} \mapsto \tau_n']$, where init contains the initial pruning constraints, each $\mathsf{incr}_{i_k}[\mathsf{inv}(o^{i_k}) \mapsto \tau_k, \bar{x}_{i_k} \mapsto \bar{c}_k]$ contains the incremental pruning constraints for one operator instance in the plan instantiated with the actual invocation timepoint and actual arguments, and $\mathsf{final}[t_{\mathsf{max}} \mapsto \tau_n']$ contains the final pruning constraints where t_{max} is instantiated according to its definition. ■

Initial pruning constraints are intended to be tested in the initial (empty) plan.

The incremental pruning constraints incr_i are intended to be tested after an instance of an operator of the corresponding type o^i has been added to the plan, generating a new search node. Incremental pruning constraints should be tested at timepoints that are relative to the invocation time of a particular action in a plan, and therefore the formal invocation timepoint $\mathsf{inv}(o^i)$ of the operator o^i may occur free in such constraints and will be substituted with the actual invocation timepoint during the planning process. Allowing different constraints to be generated for each operator type will permit certain types of control formulas to be instantiated only at the effect timepoints of a particular operator type, improving efficiency, and also serves as a preparation for the domain analysis techniques to be presented in Chapter 8, where knowledge extracted from each operator type will be used to simplify incremental pruning constraints related to that specific operator type. This is also where TALplanner will make use of the fact that the formal arguments $\overline{x}_{i_k}$ of the operator o^i may occur free in incr_i.

Final constraints are intended to be tested immediately before accepting a plan as a solution to a planning problem instance.

Note again that initial pruning constraints will be tested in the initial plan, which only contains information about the initial state. It would seem quite reasonable to assume that they would only be allowed to refer to this state, but no such conditions are present in the definitions above. On the contrary, the correctness condition for initial pruning constraints refers to these constraints holding in the *final* plan, without constraining what should hold in the initial plan, and similarly for incremental and final pruning constraints.

This is intentional. Because TALplanner correctly models incomplete knowledge about future states, limiting pruning constraints to referring to completely defined states would be unnecessarily restrictive. If an initial pruning constraint should happen to refer to the unknown future, the planner may be unable to determine at the present stage of the planning process whether or not the constraint will be guaranteed to hold in the solution plan, but this condition will always be detected – the planner will "know that it does not know". The formula will then be added to a queue of conditions that must be tested again in the future, when the relevant information may have become available.

Nevertheless, there is a cost associated with this procedure: Every time a formula is added to a queue of conditions, a pointer to the formula itself, together with a set of applicable variable bindings, must be stored in the search node to be processed again at a later stage in the search process. Therefore, in the *ideal* case, initial constraints should only evaluate fluents in the initial state, incremental constraints should only evaluate fluents in the new states generated by a newly added action, and final constraints should only ensure that the final tail of states where no more action takes place does not violate the original control formulas. This is the case for almost all pruning constraints generated in practice. This is a significant advantage over a progression framework which always, unconditionally, generates a new progressed control formula for each expanded search node – not only in theory but also in practice. As will be seen in the benchmarks in Section 6.7, the progression-based version of TALplanner requires considerably more memory than the evaluation-based version, to the extent that problems that can be solved easily with the latter planner may be unsolvable within reasonable memory limits with the former one.

Due to the desire to keep control rules stateless as much as possible, we do not anticipate being able to process all types of control formulas with optimal efficiency, but as it turns out, the full power of control rules rarely has to be utilized in most domains. Instead, almost all control rules tend to follow a fixed set of patterns that fit very well into the current formula analysis framework (Section 6.4.4). Rules that do not satisfy these patterns can still be treated in the same manner, though somewhat less efficiently. For example, if a control formula is placed in init, leaving final and all incr_i empty, the new TALplanner algorithm below will behave identically to the naive TALplanner algorithm from Definition 6.4.4 on page 170.

Adding support for incremental control formulas and condition queuing to the algorithm from Definition 6.4.4 on page 170 yields the following modified algorithm:

Definition 6.4.8 (TALplanner with Incremental Control)
Input: A goal narrative $\mathcal{N}$.
Output: A plan narrative entailing the goal $\mathcal{N}_{\mathsf{goal}}$ and the control formulas $\mathcal{N}_{\mathsf{control}}$.

```
  1  procedure TALplanner-incremental-control($\mathcal{N}$)
  2  $\gamma \leftarrow \bigwedge \mathcal{N}_{\mathsf{goal}}$                              Conjunction of all goal statements
 ▶3  $\langle \mathsf{init}, \mathsf{incr}, \mathsf{final} \rangle \leftarrow$ generate-pruning-constraints($\mathcal{N}_{\mathsf{control}}$)
 ▷4  node $\leftarrow \langle \mathsf{init}, 0, \langle\rangle \rangle$                ⟨condition queue, next invocation time, plan⟩
  5  Open $\leftarrow \langle \mathsf{node} \rangle$                                  Stack (depth first search)
  6  while Open $\neq \langle\rangle$ do
 ▷7    $\langle C, \tau, p \rangle \leftarrow$ pop(Open)                          Current plan candidate
  8    $\mathcal{N}' \leftarrow \mathcal{N} \cup p \cup$ occlude-all-after($\mathcal{N}, \tau$)        No knowledge about future
 ▶9    for all constraints $\alpha$ in $C$ do                          Check queued constraints
▶10      if $Trans^{+}(\mathcal{N}') \models Trans(\alpha)$ then $C \leftarrow C \setminus \{\alpha\}$     Remove satisfied constraint
▶11      elsif $Trans^{+}(\mathcal{N}') \models Trans(\neg\alpha)$ then backtrack           Constraint violated
▷12    $\mathcal{N}'' \leftarrow \mathcal{N} \cup p \cup \{t_{\mathsf{max}} = \tau\}$               Narrative with complete knowledge
 13    if $Trans^{+}(\mathcal{N}'') \models \mathtt{false}$ then backtrack         Consistency check (def. 6.2.12)
 14    if sequential-cycle-check($\mathcal{N}'', \tau$) then backtrack
▷15    if $Trans^{+}(\mathcal{N}'') \models Trans(\gamma \wedge C \wedge \mathsf{final})$ then   Goal + queued + final ctrl satisfied
 16      return $\mathcal{N}''$
 17    else                                               Not a solution, but check children
 18      for all actions $\mathbf{A} = [\tau, \tau']\ o^{i}(\bar{c})$ applicable over $[\tau, \tau']$ in $\mathcal{N}'$ do
▶19        $C' \leftarrow C \cup \mathsf{incr}_i[\tau, \bar{c}]$                        Old conditions + incr control
▷20        push $\langle C', \tau', \langle p; \mathbf{A} \rangle \rangle$ onto Open
 21  fail                                                                    ■
```

Compared to the previous version of the TALplanner algorithm, this algorithm is modified by splitting control rules into initial, incremental and final pruning constraints. The procedure that generates these pruning constraints will be described later in this chapter.

Whereas control rules usually refer to the complete state sequence generated by a final solution and are therefore unlikely to be entailed by intermediate search nodes, initial and incremental pruning constraints are intended to refer to limited intervals of time. Each such constraint that is applicable to a plan should therefore eventually be proven true – usually before the final solution is found but not necessarily immediately after the constraint has been posted. Consequently, the planner must keep track of which constraints have already been proven to hold in a plan and all its descendants and which constraints must be tested again when more information is available. This is done by extending the search state with a set of constraints that are not yet known to hold (the condition queue C). This set is initialized using the initial pruning constraints, and each time a new action is added, the corresponding set of incremental pruning constraints are added. Note that incremental pruning constraints are instantiated using the actual arguments and invocation timepoint of the action. Only free variables occurring in the constraints are substituted. After adding these constraints, all constraints in C – new and previously queued – are tested. If a constraint is satisfied, it can be discarded. If its negation is entailed, the planner must backtrack. If the status of the constraint cannot be determined, the constraint remains in the condition queue.

When a plan has passed the preliminary tests, the planner must test whether it is also a solution. As before, this should be done using the goal narrative $\mathcal{N}''$ where the planner assumes complete knowledge about the future. Compared to the previous version of the algorithm, there is one minor difference: The temporal constant t_{max} is set to τ, the end timepoint of the last action in the plan. This constant should only be referred to in final pruning constraints, and is intended to allow such constraints to place conditions on the final infinite tail of identical states that follows the last action in any finite plan.

Finally, if a plan is not a solution but does not appear to violate control rules, its successors should be created and pushed onto the stack of open plan candidates. This now also involves adding the incremental pruning constraints for each action to the condition queue for the corresponding search node.

6.4.4 Generating Pruning Constraints

The original control rules specified by the domain designer should be automatically analyzed in order to generate initial, incremental and final pruning constraints. Below, three common control formula classes are considered in detail.

State Constraints

A *state constraint* is a control formula $\forall t.\phi(t)$ where ϕ does not refer to states at any other time than t (that is, ϕ is a single timepoint formula for t).

For such formulas, $\phi[t \mapsto 0]$ is added to init. For each operator type o^i with formal invocation timepoint $\mathsf{inv}(o^i)$ and for each conditional or unconditional effect of this operator with temporal offset τ, the formula $\phi[t \mapsto \mathsf{inv}(o^i) + \tau]$ is added to incr_i. Nothing is added to final.

The state constraint is guaranteed to be tested in the initial state and at each timepoint where fluent values may have changed, which is equivalent to testing it at all timepoints. It is also guaranteed only to be tested at timepoints where the planner has complete knowledge, since the effect timepoints where it is tested must be within the execution interval of an action and no future effects may affect fluents in this interval given the current assumption of sequential planning.

Lemma 6.4.9
Given a single state constraint $\forall t.\phi(t)$, the analysis procedure above produces a valid set of pruning constraints $\langle \mathsf{init}, \mathsf{incr}, \mathsf{final} \rangle$. ■

Proof: We should prove that for any goal narrative $\mathcal{N}$ and plan candidate $p = \langle [\tau_1, \tau_1'] \; o_1^{i_1}(\bar{c}_1), \ldots, [\tau_n, \tau_n'] \; o_n^{i_n}(\bar{c}_n) \rangle$ for $\mathcal{N}$, the original formula $\forall t.\phi(t)$ is equivalent to $\mathsf{init} \wedge \bigwedge_{k=1}^{n} \mathsf{incr}_{i_k}[s \mapsto \tau_k, \bar{x}_{i_k} \mapsto \bar{c}_k] \wedge \mathsf{final}[t_{\mathsf{max}} \mapsto \tau_n']$, where init, incr and final are the constraints generated by the procedure above.

Assume $\forall t.\phi(t)$ holds. Any formula having the form $\phi[t \mapsto \tau]$ for some temporal term τ is an instantiation of ϕ at some specific point in time, and must also hold, since ϕ holds at all timepoints. By the construction of init, incr and final, all conjuncts in $\mathsf{init} \wedge \bigwedge_{k=1}^{n} \mathsf{incr}_{i_k}[s \mapsto \tau_k, \bar{x}_{i_k} \mapsto \bar{c}_k] \wedge \mathsf{final}[t_{\mathsf{max}} \mapsto \tau_n']$ have this form, so the conjunction also holds.

Assume $\mathsf{init} \wedge \bigwedge_{k=1}^{n} \mathsf{incr}_{i_k}[s \mapsto \tau_k, \bar{x}_{i_k} \mapsto \bar{c}_k] \wedge \mathsf{final}[t_{\mathsf{max}} \mapsto \tau_n']$ and prove that $\forall t.\phi(t)$ holds. Because the formula $\phi(t)$ is a single timepoint formula for t, its value is only affected by the state at time t and by atemporal constants. Therefore, assuming that all fluent values are identical at two timepoints τ and τ', it must necessarily be the case that $\phi(\tau)$ holds iff $\phi(\tau')$ holds. Thus, the fact that $\phi(t)$ holds at all t can be verified by testing $\phi(t)$ in all distinct states along the timeline. The state at time 0 is tested by the single constraint in init. Due to the inertia assumption, primary fluents can only change at timepoints where effects occur. Defined fluents at time t are defined by a single timepoint formula for t and can therefore also only change at timepoints where effects occur. Therefore, it is sufficient to test $\phi(t)$ at timepoints where effects occur. The construction of incr guarantees that this happens.

Note that an interval effect for the interval $[\tau, \tau']$ only generates one instantiation of the state constraint, at time τ. Only state transitions are relevant, and because interval effects force a single fluent to take on the same value throughout the entire interval, the only potential state transition caused by an interval effect is at the beginning of the interval.

Triggered State Transition Constraints

A *triggered state transition constraint* is a control formula $\forall t.\phi(t)$ where ϕ only refers to states in $[t, t+1]$ and where ϕ written in disjunctive normal form includes the two disjuncts $[t]\ f \hat{=} \omega$ and $[t+1]\ f \hat{=} \omega'$, where f is a fluent and ω and ω' are value terms that cannot take on the same value. This type of constraint is quite common – all constraints in Example 6.4.1 on page 168 are of this type – and can only be false if the fluent f changes values from t to $t+1$.

For such formulas, nothing is added to init. For each operator type o^i with formal invocation timepoint $\mathsf{inv}(o^i)$ and for each conditional or unconditional effect of this operator with temporal offset τ, the formula $\phi[t \mapsto \mathsf{inv}(o^i) + \tau - 1]$ is added to incr_i. Because τ is guaranteed to be at least 1, the temporal term $\mathsf{inv}(o^i) + \tau - 1$ is well-defined even with a restriction to non-negative time. Nothing is added to final.

Lemma 6.4.10
Given a single triggered state transition constraint $\forall t.\phi(t)$, the analysis procedure above produces a valid set of pruning constraints $\langle \mathsf{init}, \mathsf{incr}, \mathsf{final}\rangle$. ■

Proof: We should prove that for any goal narrative $\mathcal{N}$ and plan candidate $p = \langle [\tau_1, \tau_1']\ o_1^{i_1}(\bar{c}_1), \ldots, [\tau_n, \tau_n']\ o_n^{i_n}(\bar{c}_n)\rangle$ for $\mathcal{N}$, the original formula $\forall t.\phi(t)$ is equivalent to $\mathsf{init} \wedge \bigwedge_{k=1}^{n} \mathsf{incr}_{i_k}[s \mapsto \tau_k, \bar{x}_{i_k} \mapsto \bar{c}_k] \wedge \mathsf{final}[t_{\mathsf{max}} \mapsto \tau_n']$, where init, incr and final are the constraints generated by the procedure above.

Assume $\forall t.\phi(t)$ holds. Any formula having the form $\phi[t \mapsto \tau]$ for some τ is an instantiation of ϕ at some specific point in time, and must also hold, since ϕ holds at all timepoints. By the construction of init, incr and final, all conjuncts in $\mathsf{init} \wedge \bigwedge_{k=1}^{n} \mathsf{incr}_{i_k}[s \mapsto \tau_k, \bar{x}_{i_k} \mapsto \bar{c}_k] \wedge \mathsf{final}[t_{\mathsf{max}} \mapsto \tau_n']$ have this form, so the conjunction also holds.

Assume $\mathsf{init} \wedge \bigwedge_{k=1}^{n} \mathsf{incr}_{i_k}[s \mapsto \tau_k, \bar{x}_{i_k} \mapsto \bar{c}_k] \wedge \mathsf{final}[t_{\mathsf{max}} \mapsto \tau_n']$ and prove that $\forall t.\phi(t)$ holds. By the definition of this type of state transition constraint, $\phi(t)$ can only be false if there is a change in the value of a specific fluent from time $t-1$ to time t. Due to the inertia assumption, primary fluents can only change at timepoints where effects occur. Defined fluents at time t are defined by a single timepoint formula for t and can therefore also only change at timepoints where effects occur. Therefore, it is sufficient to test $\phi(t)$ at $t = \tau - 1$ for each effect taking place at time τ. The construction of incr guarantees that this happens.

Example 6.4.2 (Logistics Domain, continued)
The six operators of the standard logistics domain were defined in Example 6.2.10 on page 157. All operators share the constant duration 1 and use the formal invocation timepoint $\mathsf{inv}(o^i) = s$.

The three control rules for the logistics domain that were shown in Example 6.4.1 on page 168 are examples of triggered state transition constraints.

For only-load-into-plane-when-necessary, the following formula is added to each incr_i:

$\forall obj, plane, loc.$
$[s]\ \neg\mathsf{in}(obj, plane) \wedge \mathsf{at}(obj, loc) \wedge [s{+}1]\ \mathsf{in}(obj, plane) \rightarrow$
$\exists loc'\ [\ \mathsf{goal}(\mathsf{at}(obj, loc')) \wedge [s]\ \mathsf{city\text{-}of}(loc) \not\hat{=} \mathsf{city\text{-}of}(loc')\ ]$

For only-unload-from-plane-when-necessary, the following formula is added to each incr_i:

$\forall obj, plane, loc.$
$[s]\ \mathsf{in}(obj, plane) \wedge \mathsf{at}(plane, loc) \wedge [s{+}1]\ \neg\mathsf{in}(obj, plane) \rightarrow$
$\exists loc'\ [\ \mathsf{goal}(\mathsf{at}(obj, loc')) \wedge [s]\ \mathsf{city\text{-}of}(loc) \hat{=} \mathsf{city\text{-}of}(loc')\ ]$

For objects-remain-at-destinations, the following formula is added to each incr_i:

$\forall obj, loc.$
$[s]\ \mathsf{at}(obj, loc) \wedge \mathsf{goal}(\mathsf{at}(obj, loc)) \rightarrow [s{+}1]\ \mathsf{at}(obj, loc)$ ■

State Transition Constraints

A *state transition constraint* is a control formula $\forall t.\phi(t)$ where ϕ only refers to states in $[t, t+1]$. For such formulas, $\phi[t \mapsto 0]$ is added to init. For each operator type o^i with formal invocation timepoint $\mathsf{inv}(o^i)$ and for each conditional or unconditional effect of this operator with temporal offset τ, the formula $\phi[t \mapsto \mathsf{inv}(o^i) + \tau] \wedge \phi[t \mapsto \mathsf{inv}(o^i) + \tau - 1]$ is added to incr_i. Nothing is added to final.

Lemma 6.4.11
Given a single state transition constraint $\forall t.\phi(t)$, the analysis procedure above produces a valid set of pruning constraints $\langle \mathsf{init}, \mathsf{incr}, \mathsf{final} \rangle$. ■

Proof: We should prove that for any goal narrative $\mathcal{N}$ and plan candidate $p = \langle [\tau_1, \tau_1']\ o_1^{i_1}(\bar{c}_1), \ldots, [\tau_n, \tau_n']\ o_n^{i_n}(\bar{c}_n) \rangle$ for $\mathcal{N}$, the original formula $\forall t.\phi(t)$ is equivalent to $\mathsf{init} \wedge \bigwedge_{k=1}^{n} \mathsf{incr}_{i_k}[s \mapsto \tau_k, \bar{x}_{i_k} \mapsto \bar{c}_k] \wedge \mathsf{final}[t_{\mathsf{max}} \mapsto \tau_n']$, where init, incr and final are the constraints generated by the procedure above.

Assume $\forall t.\phi(t)$ holds. Any formula having the form $\phi[t \mapsto \tau]$ for some τ is an instantiation of ϕ at some specific point in time, and must also hold, since ϕ holds at all timepoints. By the construction of init, incr and final, all conjuncts in $\mathsf{init} \wedge \bigwedge_{k=1}^{n} \mathsf{incr}_{i_k}[s \mapsto \tau_k, \bar{x}_{i_k} \mapsto \bar{c}_k] \wedge \mathsf{final}[t_{\mathsf{max}} \mapsto \tau_n']$ have this form, so the conjunction also holds.

Assume $\mathsf{init} \wedge \bigwedge_{k=1}^{n} \mathsf{incr}_{i_k}[s \mapsto \tau_k, \bar{x}_{i_k} \mapsto \bar{c}_k] \wedge \mathsf{final}[t_{\mathsf{max}} \mapsto \tau_n']$ and prove that $\forall t.\phi(t)$ holds. By the definition of state transition constraints, the formula $\phi(t)$ is only affected by the states at times t and $t+1$ and by atemporal constants. Therefore, assuming that all fluent values at times τ and τ' are identical, and that all fluent values at times $\tau + 1$ and $\tau' + 1$ are identical, it must necessarily be the case that $\phi(\tau) \equiv \phi(\tau')$. Thus, the fact that $\phi(t)$ holds at all t can be verified by testing $\phi(t)$ in all distinct successive state pairs $[\tau, \tau+1]$ along the timeline. The state pair at time $[0, 1]$ is tested by the single constraint in init. Given that no fluents ever change, this is the only distinct successive state pair. A change in fluent values at time τ gives rise to two potential new distinct successive state pairs at $[\tau - 1, \tau]$ and $[\tau, \tau + 1]$. Due to the inertia assumption, primary fluents can only change at timepoints where effects occur. Defined fluents at time t are defined by a single timepoint formula for t and can therefore also only change at timepoints where effects occur. Therefore, it is sufficient to test $\phi(t)$ at $t = \tau - 1$ and $t = \tau$ for each effect taking place at time τ. The construction of incr guarantees that this happens.

Additional Control Formula Classes

The class of state transition constraints can trivially be extended to formulas of the form $\forall t.\phi(t)$, where ϕ only refers to states in $[t, t + k]$ for some constant k, by increasing the number of instantiations of the formula ϕ.

It is also possible to treat control formulas of the form $\forall t.\phi(t)$ where ϕ only refers to states in $[t, t + d]$ and d is a non-constant temporal term independent of t. The formula $d = 0 \rightarrow \phi[t \mapsto 0]$ is added to init. For each operator type o^i with duration δ, the formula $\forall k.1 \leq k \leq \delta \rightarrow \phi[t \mapsto \mathsf{inv}(o^i) + k - d]$ should be added to incr_i, but since $\mathsf{inv}(o^i) + k - d$ could be negative and TAL currently uses non-negative time, the formula has to be rewritten as $\forall k.1 \leq k \leq \delta \rightarrow (\forall t.t + d = \mathsf{inv}(o^i) + k \rightarrow \phi(t))$. Finally, $\forall k.1 \leq k \leq d \rightarrow (\forall t.t + d = t_{\mathsf{max}} + k \rightarrow \phi(t))$ is added to final.

6.5 Tense Control Rules and Progression

In addition to supporting standard TAL control formulas as defined in the previous section, the domain modeling language $\mathcal{L}(\mathrm{ND})^*$ has also been extended with a set of new tense macros emulating the modal tense operators used in TLPlan. Formulas written using the new tense macros can be used together with a modified progression algorithm. This was intended to improve the analysis of TALplanner performance relative to TLPlan performance: If both TLPlan and TALplanner use tense control formulas, most differences in performance should be due to differences in lower level algorithms, while the performance difference between TALplanner using tense control formulas and TALplanner using TAL-based control formulas must be due to the difference between using a progression algorithm or a formula evaluation method as described in the previous section. Furthermore, while the formula evaluation method seemed promising, there were some less common types of control formula that did not fit easily into the separation into initial, incremental and final control, most notably those formulas using the "until" operator. Supporting both types of control formula enables the user to choose the most suitable type for the task at hand.

The tense operators that were added to $\mathcal{L}(\mathrm{ND})^*$ were not adapted from LTL, but from MITL (Metric Interval Temporal Logic, Alur, Feder, & Henzinger, 1991, Alur & Henzinger, 1992), which was also used by Bacchus and Kabanza (1998) in the context of planning for temporally extended goals. Like LTL, MITL supports the three temporal operators U (until), $\Diamond$ (eventually), and $\Box$ (always), the main difference being that the operators can be indexed with closed, open, or semi-open temporal intervals. For example, it is possible to state that a formula must eventually become true *within a certain interval*. MITL was a better match for TALplanner than LTL, given TALplanner's support for operators with extended duration.

Though MITL does not support the $\bigcirc$ (next) operator, due to its use of dense time, this operator can still be used in TALplanner which uses discrete time.

In previous publications, the operators added to $\mathcal{L}(\text{ND})^*$ were called "modal operators", and TALplanner control formulas using these operators have been described as "modal control formulas". This is directly misleading, since TAL is not a modal logic. Since the operators are tense-based rather than based on explicit time, we now call the operators "tense operators", and control formulas written using tense operators are called "tense formulas", even though this is admittedly still not a perfect name given that these terms are often used in modal tense logics.

Definition 6.5.1 (Tense Formula)
A *tense formula* in $\mathcal{L}(\text{ND})^*$ is one of the following:

- A temporal formula, value formula, or fluent formula (Section 2.3.3 on page 20).
- A goal expression (Section 6.2.6 on page 150).
- $\phi \, \mathsf{U}_{[\tau,\tau']} \, \psi$, where ϕ and ψ are tense formulas and τ and τ' are temporal terms.
- $\Diamond_{[\tau,\tau']} \, \phi$, where ϕ is a tense formula and τ and τ' are temporal terms.
- $\Box_{[\tau,\tau']} \, \phi$, where ϕ is a tense formula and τ and τ' are temporal terms.
- $\bigcirc \phi$, where ϕ is a tense formula.
- A combination of tense formulas using the standard logical connectives and quantification over values. ■

We will also use the shorthand notation $\phi \, \mathsf{U} \, \psi \equiv \phi \, \mathsf{U}_{[0,\infty]} \, \psi$, $\Diamond \phi \equiv \Diamond_{[0,\infty]} \, \phi$, and $\Box \phi \equiv \Box_{[0,\infty]} \, \phi$.

Definition 6.5.2 (Tense Control Statement)
A *tense control statement* in $\mathcal{L}(\text{ND})^*$, labeled **tcontrol**, consists of a tense formula. ■

Note that while ordinary TAL formulas use absolute time, MITL formulas (and therefore tense formulas) use relative time, where a formula only has a value relative to a "current state". The meaning of a formula containing a temporal operator is therefore dependent on the timepoint at which it is evaluated, as illustrated by the following example.

Example 6.5.1 (Relative and Absolute Time)
The formulas $\forall t.[t]\phi \rightarrow [t, t+5]\ \psi$ and $\Box(\phi \rightarrow \Box_{[0,5]}\ \psi)$ have the same meaning: At any timepoint where ϕ holds, ψ must hold until five timepoints later. Notice the difference in the specification of the temporal interval: Ordinary TAL uses absolute time, where the starting timepoint t in the interval $[t, t+5]$ must be explicitly specified, while the interval $[0, 5]$ specified in the tense macro is interpreted relative to the time when ϕ was true.

The formula $\forall t.[t]\phi \rightarrow [0, 5]\ \psi$, on the other hand, means that for each timepoint t where ϕ holds, ψ must hold at the absolute interval $[0, 5]$, even though this may be before t. This formula could also be written $(\exists t.[t]\ \phi) \rightarrow [0, 5]\ \psi$. ■

For this reason, reducing tense operators to $\mathcal{L}(\text{FL})$ requires a temporal context. The following translation function can be used to translate tense formulas into $\mathcal{L}(\text{ND})^*$ without tense operators and further into $\mathcal{L}(\text{FL})$.

Definition 6.5.3 (Translation of Tense Formulas)
Let $\overline{\tau}$ be a temporal term and γ be a tense control formula intended to be evaluated at $\overline{\tau}$. Then, the following procedure returns an equivalent formula in $\mathcal{L}(\text{ND})^*$ without tense operators. In the following, $\mathcal{Q}$ denotes a quantifier and $\otimes$ denotes a binary logical connective.

```
1  procedure TransTense($\overline{\tau}, \gamma$)
2  if $\gamma = \mathcal{Q}x.\phi$ then return $\mathcal{Q}x.$TransTense($\overline{\tau}, \phi$)
3  if $\gamma = \phi \otimes \psi$ then return TransTense($\overline{\tau}, \phi$) $\otimes$ TransTense($\overline{\tau}, \psi$)
4  if $\gamma = \neg\phi$ then return $\neg$TransTense($\overline{\tau}, \phi$)
5  if $\gamma = f(\overline{x}) \hat{=} v$ then return $[\overline{\tau}]\ \gamma$
6  if $\gamma$ contains no tense operator then return $\gamma$
7  if $\gamma = \phi \mathsf{U}_{[\tau,\tau']}\ \psi$ then return
   $\exists t[\overline{\tau} + \tau \leq t \wedge t \leq \overline{\tau} + \tau' \wedge$
       TransTense($t, \psi$) $\wedge \forall t'[\overline{\tau} \leq t' \wedge t' < t \rightarrow$ TransTense($t', \phi$)]]
8  if $\gamma = \circ\phi$ then return TransTense($\overline{\tau} + 1, \phi$)
9  if $\gamma = \Box_{[\tau,\tau']}\ \psi$ then return $\forall t[\overline{\tau} + \tau \leq t \wedge t \leq \overline{\tau} + \tau' \rightarrow$ TransTense($t, \phi$)]
10 if $\gamma = \Diamond_{[\tau,\tau']}\ \psi$ then return $\exists t[\overline{\tau} + \tau \leq t \wedge t \leq \overline{\tau} + \tau' \wedge$ TransTense($t, \phi$)]
```

The *Trans* translation function is extended for tense control formulas by defining *Trans*(γ) = *Trans*(TransTense$(0, \gamma)$). ■

During the planning process, tense control formulas are not translated into $\mathcal{L}(\text{FL})$. Instead, they are progressed through states using the Progress algorithm. This algorithm is similar to the one used by Bacchus and Kabanza (1998) for progressing formulas in a first-order version of MITL (metric interval temporal logic). However, there are some differences in the progression of the tense operators, since TAL actions with duration can have internal state, with a sequence of state changes between the initiation state and the effect state.

The progression algorithm below satisfies the following property. Let ϕ be a tense formula, τ and τ' two numeric timepoints such that $\tau < \tau'$, and $\mathcal{N}$ an $\mathcal{L}(\text{ND})^*$ narrative. Then, ϕ will hold at τ in $\mathcal{N}$ iff $\mathsf{Progress}(\phi,\tau,\tau',\mathcal{N})$ holds at τ' in $\mathcal{N}$. More formally, $\mathit{Trans}^+(\mathcal{N}) \models \mathit{Trans}(\mathsf{TransTense}(\tau,\phi))$ iff $\mathit{Trans}^+(\mathcal{N}) \models \mathit{Trans}(\mathsf{TransTense}(\tau',\mathsf{Progress}(\phi,\tau,\tau',\mathcal{N})))$.

Definition 6.5.4 (Progression of Tense Formulas)
The following algorithm is used for progression of tense control formulas in TAL-planner.

```
procedure $\mathsf{Progress}(\phi,\tau,\tau',\mathcal{N})$
if $\tau = \tau'$ return $\phi$
if $\phi = f(\overline{x}) \hat{=} v$
  if $\mathit{Trans}^+(\mathcal{N}) \models \mathit{Trans}([\tau]\ \phi)$ return true else return false
if $\phi = \neg\phi_1$ return $\neg\mathsf{Progress}(\phi_1,\tau,\tau',\mathcal{N})$
if $\phi = \phi_1 \otimes \phi_2$ return $\mathsf{Progress}(\phi_1,\tau,\tau',\mathcal{N}) \otimes \mathsf{Progress}(\phi_2,\tau,\tau',\mathcal{N})$
if $\phi = \forall x.\phi$ // where $x$ belongs to the finite TAL value domain $X$
  return $\bigwedge_{c\in X} \mathsf{Progress}(\phi[x \mapsto c],\tau,\tau',\mathcal{N})$
if $\phi = \exists x.\phi$ // where $x$ belongs to the finite TAL value domain $X$
  return $\bigvee_{c\in X} \mathsf{Progress}(\phi[x \mapsto c],\tau,\tau',\mathcal{N})$
if $\phi$ contains no tense operator
  if $\mathit{Trans}^+(\mathcal{N}) \models \mathit{Trans}(\phi)$ return true else return false
if $\phi = \phi_1 \mathsf{U}_{[\tau_1,\tau_2]} \phi_2$
  if $[\tau_1,\tau_2] < 0$ return false
  elsif $0 \in [\tau_1,\tau_2]$ return $\mathsf{Progress}(\phi_2,\tau,\tau',\mathcal{N}) \vee$
            $(\mathsf{Progress}(\phi_1,\tau,\tau',\mathcal{N}) \wedge \mathsf{Progress}(\phi_1 \mathsf{U}_{[\tau_1-1,\tau_2-1]} \phi_2,\tau+1,\tau',\mathcal{N}))$
  else return $\mathsf{Progress}(\phi_1,\tau,\tau',\mathcal{N}) \wedge$
                 $\mathsf{Progress}(\phi_1 \mathsf{U}_{[\tau_1-1,\tau_2-1]} \phi_2,\tau+1,\tau',\mathcal{N})$
if $\phi = \Diamond_{[\tau_1,\tau_2]}\ \phi_1$
  if $[\tau_1,\tau_2] < 0$ return false
  elsif $0 \in [\tau_1,\tau_2]$ return $\mathsf{Progress}(\phi_1,\tau,\tau',\mathcal{N}) \vee$
                                 $\mathsf{Progress}(\Diamond_{[\tau_1-1,\tau_2-1]}\ \phi_1,\tau+1,\tau',\mathcal{N})$
  else return $\mathsf{Progress}(\Diamond_{[\tau_1-1,\tau_2-1]}\ \phi_1,\tau+1,\tau',\mathcal{N})$
if $\phi = \Box_{[\tau_1,\tau_2]}\ \phi_1$
  if $[\tau_1,\tau_2] < 0$ return true
  elsif $0 \in [\tau_1,\tau_2]$ return $\mathsf{Progress}(\phi_1,\tau,\tau',\mathcal{N}) \wedge$
                                 $\mathsf{Progress}(\Box_{[\tau_1-1,\tau_2-1]}\ \phi_1,\tau+1,\tau',\mathcal{N})$
  else return $\mathsf{Progress}(\Box_{[\tau_1-1,\tau_2-1]}\ \phi_1,\tau+1,\tau',\mathcal{N})$
if $\phi = \bigcirc\,\phi_1$
  if $\tau + 1 = \tau'$ return $\phi_1$
  else return $\mathsf{Progress}(\phi_1,\tau+1,\tau',\mathcal{N})$
```

The result of Progress is simplified using the rules $\neg\texttt{false} = \texttt{true}$, $(\texttt{false} \wedge \alpha) = (\alpha \wedge \texttt{false}) = \texttt{false}$, $(\texttt{false} \vee \alpha) = (\alpha \vee \texttt{false}) = \alpha$, $\neg\texttt{true} = \texttt{false}$, $(\texttt{true} \wedge \alpha) = (\alpha \wedge \texttt{true}) = \alpha$, and $(\texttt{true} \vee \alpha) = (\alpha \vee \texttt{true}) = \texttt{true}$. ■

Implementation note. For planning domains where actions have long durations, there may be extended periods of time where no changes occur. Because TAL associates a state with each integer timepoint, this leads to extended sequences of identical states. The TALplanner implementation uses a slightly modified progression algorithm which can progress some formulas through such sequences of identical states in a single step, while retaining the same results as the algorithm above where the progression of $\Box$, $\Diamond$ and U always proceeds one discrete timepoint at a time.

6.5.1 The TALplanner Algorithm with Tense Control Rules

Adding support for tense control rules to TALplanner yields the following algorithm, described below.

Definition 6.5.5 (TALplanner with Incremental and Tense Control)
Input: A goal narrative $\mathcal{N}$.
Output: A plan narrative entailing the goal $\mathcal{N}_{\mathsf{goal}}$ and the control formulas $\mathcal{N}_{\mathsf{control}}$.

```
  1  procedure TALplanner-tense-control($\mathcal{N}$)
  2  $\gamma \leftarrow \bigwedge \mathcal{N}_{\mathsf{goal}}$                                  Conjunction of all goal statements
 ▶3  $\mu \leftarrow \bigwedge \mathcal{N}_{\mathsf{tcontrol}}$                               Conjunction of all tense control rules
  4  $\langle \mathsf{init}, \mathsf{incr}, \mathsf{final} \rangle \leftarrow \mathsf{generate\text{-}pruning\text{-}constraints}(\mathcal{N}_{\mathsf{control}})$
 ▷5  $\mathsf{node} \leftarrow \langle \mu, \mathsf{init}, -1, 0, \langle\rangle \rangle$       ⟨control, cond. queue, last inv. time, next inv. time, plan⟩
  6  $\mathsf{Open} \leftarrow \langle \mathsf{node} \rangle$                                   Stack (depth first search)
  7  while $\mathsf{Open} \neq \langle\rangle$ do
 ▷8    $\langle \mu, C, \tau_0, \tau, p \rangle \leftarrow \mathbf{pop}(\mathsf{Open})$                  Current plan candidate
  9    $\mathcal{N}' \leftarrow \mathcal{N} \cup p \cup \mathsf{occlude\text{-}all\text{-}after}(\mathcal{N}, \tau)$      No knowledge about future
 10    for all constraints $\alpha$ in $C$ do                           Check queued constraints
 11      if $Trans^{+}(\mathcal{N}') \models Trans(\alpha)$ then $C \leftarrow C \setminus \{\alpha\}$     Remove satisfied constraint
 12      elsif $Trans^{+}(\mathcal{N}') \models Trans(\neg\alpha)$ then backtrack        Constraint violated
▶13    $\mu^{+} \leftarrow \mathsf{Progress}(\mu, \tau_0 + 1, \tau + 1, \mathcal{N}')$                 Progress tense control
▶14    if $\mu^{+} = \mathtt{false}$ then backtrack
 15    $\mathcal{N}'' \leftarrow \mathcal{N} \cup p \cup \{t_{\mathsf{max}} = \tau\}$               Narrative with complete knowledge
 16    if $Trans^{+}(\mathcal{N}'') \models \mathtt{false}$ then backtrack        Consistency check (def. 6.2.12)
 17    if $\mathsf{sequential\text{-}cycle\text{-}check}(\mathcal{N}'', \tau)$ then backtrack
 18    if $Trans^{+}(\mathcal{N}'') \models Trans(\gamma \wedge C \wedge \mathsf{final})$ then   Goal + queued + final ctrl satisfied
 19      return $\mathcal{N}''$
 20    else                                            Not a solution, but check children
 21      for all actions $\mathbf{A} = [\tau, \tau']\ o^{i}(\bar{c})$ applicable over $[\tau, \tau']$ in $\mathcal{N}'$ do
 22        $C' \leftarrow C \cup \mathsf{incr}_i[\tau, \bar{c}]$                         Old conditions + incr control
▷23        push $\langle \mu^{+}, C', \tau, \tau', \langle p; \mathbf{A} \rangle \rangle$ onto Open
 24  fail                                                                  ■
```

Each search node is associated with a tense control formula μ, which may be the constant `true` in the absence of tense control. In order to determine the temporal interval over which the tense control formula should be progressed in each search node, it is also necessary to include the last action invocation timepoint in the search node. This timepoint is denoted by τ_0 in the algorithm, while the next action invocation timepoint (or equivalently, the end timepoint of the last action) is still denoted by τ.

Notice that the call to the progression algorithm specifies the temporal parameters $\tau_0 + 1$ and $\tau + 1$. This is interpreted as a request for a new formula μ^+ that should hold at $\tau + 1$ iff the original formula μ holds at $\tau_0 + 1$, thereby causing a progression through the semi-open temporal interval $[\tau_0 + 1, \tau + 1)$. Because discrete integer time is used, this is equivalent to progression through all states in the closed interval $[\tau_0 + 1, \tau]$, corresponding to the new fixed state or states generated by the most recently added action in the current plan. If $\mu^+ = \texttt{false}$, the search node should be pruned.

Notice that in the initial node, $\tau_0 = -1$ and $\tau = 0$. This provides the correct boundary condition for progression through the initial state: $[\tau_0 + 1, \tau] = [0, 0]$.

6.6 Completeness, Control and the Definition of Plans

A planner is *complete* iff it is guaranteed to return a solution for every problem instance except those for which no solution exists. At the surface this concept appears quite simple, but a deeper investigation of completeness reveals some subtleties that are not necessarily apparent at first glance.

It is quite easy to fall into the trap of relying on a subconscious intuitive picture of what it means that "no solution exists", a picture that most likely involves what is actually possible to do in a real world scenario. After all, if it is logically or physically impossible to achieve a certain set of goals, even a complete planner could not be expected to come up with a solution! But this is not what was stated in the definition of completeness. In this definition, a "solution" is a formally defined concept meaning approximately "plan that ensures the specified goals are satisfied". This leads us to "plan", which is also a formally defined concept – a concept over which we have a significant degree of control. In some cases, a planning algorithm can be made complete or incomplete by the simple expedient of altering the definition of "plan", without stepping outside the bounds of what would on the surface appear to be a reasonable definition.

Similarly, completeness is only defined relative to the set of domains that can be modeled in the input language for a planner and relative to the aspects of those domains that can be modeled. Altering the input language can restrict or expand the set of planning domains and problem instances to which the planner can be applied, again affecting the completeness of the planning algorithm.

TALplanner illustrates both of these points.

Though TALplanner could in theory use any forward-chaining search algorithm, all versions of the planner presented in this thesis use plain depth first search, which is only complete if all branches are finite.

If plans were defined as arbitrary executable sequences of action occurrences, the search tree would contain branches with infinitely many nodes, for two reasons. First, in any given state there may be an applicable action whose effects are reversible, leading back to the same state, where it must be possible to apply the same action once again, ad infinitum. For example, picking up an item and immediately putting it down must lead to a state where the item can once more be picked up, put down, picked up, and so on, generating an infinitely long branch in the search tree. Second, action occurrences in TALplanner are timed. Even though there is a finite number of ground action instances, these action instances could theoretically be applied at arbitrary delays from the previous action, generating infinitely many children for every search node.

One potential solution to this problem can be found in changing the definition of "plan", stating that a plan is an executable sequence of action occurrences that does not contain cycles or temporal gaps. These added constraints on valid plans are sufficient to make TALplanner complete, even when depth first search is used. But is this altered definition reasonable?

The first version of TALplanner presented in this chapter had no support for control rules, searching aimlessly through the forward-chaining search tree for a plan achieving the intended goals (Definition 6.3.1 on page 164). For this algorithm, the new definition of "plan" should be completely non-controversial. After all, there is definitely no good reason to generate plans with cycles when for any cyclic solution there must necessarily be a corresponding shorter (and in most conceivable circumstances better) acyclic solution. Temporal gaps in the execution of a plan can also be forbidden, because for any solution containing a temporal gap there must necessarily be a shorter solution without gaps generated by moving all action occurrences as early as possible while still retaining the original action order. Thus, changing the definition of "plan" and making a few minor adjustments to the search tree is sufficient to make the planner complete.

The addition of control rules in Definition 6.4.4 on page 170 is a fundamental change affecting the planner at more levels than is immediately apparent. In this discussion, the most salient aspect of this change is the fact that unlike ordinary state goals, which only constrain the final state achieved by a solution plan, control rules can place arbitrary constraints on the entire state sequence generated by the solution. This enables the construction of problem instances whose solutions *must* contain temporal gaps and *must* contain state cycles – but our new definition of "plan" would prevent such solutions from being explored. There are two different ways of viewing this conflict.

First, we can view acyclicity and the lack of temporal gaps as simple optimizations that were merely introduced in order to avoid visiting parts of the search space only containing redundant solutions. In this case, these optimizations are obviously invalidated by the introduction of control rules, and despite being introduced to achieve completeness, they could in themselves be the cause of incompleteness. In other words, because new aspects of a planning domain can be modeled, the planner is now incomplete.

Second, acyclicity and the lack of temporal gaps can be viewed as essential features of our concept of a plan. In this case, the ability to construct a control rule that is only satisfied by plans with temporal gaps is seen as a trivial consequence of the expressive power of control rules. After all, control rules also let us express logically impossible conditions such as "at some point in time, false should be true" or conditions such as "all goals should be achieved at time 1" which may be physically impossible to achieve for any given problem instance, and a planner should not be seen as incomplete for not being able to construct plans satisfying these rules.

Thus, whether or not TALplanner is complete may be said to depend on the reader's concept of a plan.

6.7 Evaluation vs Progression: Initial Benchmark Tests

It is not immediately obvious whether evaluation-based or progression-based control rules would provide the best performance in TALplanner. Whereas control based on progression has certain advantages in terms of guaranteeing that no part of a control formula is evaluated twice in the same state, it also has the disadvantage of requiring the construction of new tense control formulas for each search node, which increases memory usage as well as the amount of time spent on memory management and object construction. Nevertheless, the time requirements for the two methods ought to remain within a constant factor of each other, with no difference in time complexity. This hypothesis was tested for an early version of TALplanner using the standard logistics domain and blocks world.

We also compared TALplanner with tense control rules to TLPlan using the same planning domains. Because the planners at this early stage used identical search procedures and because identical control rules were used, the same plans were generated by the two planners and any differences in performance must be due to different lower-level algorithms and data structures. These results will not be analyzed further in this chapter, but will serve to separate timing differences caused by implementation issues from the performance impact of formula analysis and optimization as described in Chapter 8.

Logistics. For the logistics domain, we tested 30 problem instances from the First International Planning Competition (IPC-1998, McDermott, 1998). The results are shown in Table 6.1. The *Ops* column shows the length of each solution. The *Nodes* column contains the number of search nodes that were examined by TALplanner, which in this early version was identical to the number of states that were created by TLPlan. The remaining columns show times (in seconds) for TLPlan, TALplan-

Number	Ops	Nodes	TLPlan	TALplanner tense	TALplanner TAL
1	26	586	0.421	0.190	0.160
2	33	1665	1.712	0.541	0.501
3	55	4809	19.398	2.934	3.505
4	59	8019	54.338	5.528	7.581
5	22	494	0.310	0.281	0.250
6	72	8840	84.191	14.320	19.919
7	34	2652	5.568	1.101	1.032
8	41	6407	97.310	11.726	14.241
9	85	16308	218.644	15.102	24.405
10	105	13025	167.581	14.321	18.436
11	31	2250	5.167	0.961	0.872
12	41	16614	286.021	23.464	41.840
13	67	23548	1073.263	35.071	50.753
14	94	17606	802.824	24.826	36.282
15	94	5048	24.675	3.154	3.135
16	58	14243	168.002	7.961	10.966
17	45	7196	90.460	5.278	6.850
18	170	62445	4358.367	69.299	104.009
19	153	46884	2685.021	35.872	57.112
20	150	64984	3414.089	73.176	141.984
21	104	51857	2102.643	50.974	95.597
22	296	130937		632.730	1027.097
23	115	9744	116.798	8.232	9.954
24	41	27215	695.780	22.513	47.158
25	190	133137	11724.910	318.427	680.428
26	194	46284	9976.946	427.955	699.576
27	149	75907	14994.551	265.411	463.737
28	274	399213		2093.170	4613.333
29	330	132331	60874.834	353.739	545.845
30	136	130191	14070.923	440.373	1036.160

Table 6.1: Initial Logistics Test Results

ner using tense control and progression, and TALplanner using TAL control and formula evaluation.

Blocks World. For the standard blocks world, we found no “standard” problem instances large enough to truly challenge TLPlan or TALplanner, and therefore manually created a number of different instances using between 25 and 1000 blocks. In retrospect, it would have been better to use a larger number of problem instances with a more random distribution of blocks. However, at the moment these tests were performed, many efficiency improvements were planned and doing an exhaustive test of the current version of TALplanner would perhaps not have been

Blocks	Ops	Nodes	TLPlan		TALplan/tense		TALplan/TAL	
			time	mem	time	mem	time	mem
25	16	344	0.110	3104	0.110	6640	0.060	6612
50	70	2295	1.963	5672	1.603	6792	1.302	6644
70	88	4109	5.868		3.916		3.715	
70	106	4361	7.501	8752	4.677	7516	4.406	6644
100	160	8945	37.254	14912	14.441	7772	14.060	6644
140	232	17829	185.497	27532	39.246	9372	41.940	7208
200	405	35704	919.262		118.410		142.996	
280	405	68104	3756.061		277.799		315.654	
280	580	74297	4297.750	104464	394.196	21308	474.012	8536
460	580	178697	32303.100	178884	3208.159	39528	1899.992	9840
460	904	187607			1941.521		2535.946	
640	1228	365069			5862.197	68464	7679.733	14284
820	1908	463779			10487.159	95620	12837.629	18732
1000	2232	718281					25028.509	24264

Table 6.2: Initial Blocks World Test Results

the best use of our time.

The results for the blocks world were first presented in Doherty and Kvarnström (1999). Lack of space in this preliminary report forced us to report only a representative subset of our test results. Table 6.2 now contains the previously published results as well as the results that were omitted. The *Blocks* column shows the number of blocks for each problem instance. As before, the *Ops* column shows the length of each solution and the *Nodes* column contains the number of search nodes that were examined by TALplanner. The remaining columns show times (in seconds) and memory usage (in kilobytes) for TLPlan, TALplanner using tense control and progression, and TALplanner using TAL control and formula evaluation. Unfortunately, memory usage measurements are not available for the test results that were omitted from Doherty and Kvarnström (1999).

Result Analysis. We can immediately see that TALplanner requires considerably less memory than TLPlan and that evaluation requires considerably less memory than progression, allowing significantly larger problem instances to be handled before the available resources are exhausted. It is also quite obvious that TALplanner was considerably faster than TLPlan for these two domains, regardless of whether progression or evaluation is used.

For the problem instances we tested, progression outperforms formula evaluation in most cases, but generally only by a factor of 2 to 2.5 for the logistics domain and by a factor of 1.2 to 1.3 for the blocks world. Apparently, the overhead for memory management and object construction in the progression algorithm does not completely negate the advantage of never evaluating a formula twice in the same state.

Though this is by no means a complete statistical analysis and additional domains could certainly have been used to strengthen these claims, it would nevertheless appear reasonably safe to say that there is only a constant factor difference in performance between the two approaches, especially because both progression and formula evaluation do have to evaluate approximately the same number of fluents in the state sequence generated by a plan.

The current performance advantage for the progression algorithm does not imply that formula evaluation should be avoided. In addition to the more modest memory requirements for formula evaluation as demonstrated by these benchmark tests, formula evaluation also lends itself more easily to certain optimization techniques which can improve the performance of the planner by orders of magnitude in some cases (Chapter 8).

6.7.1 Program Versions and Test Procedures

The computer used for the tests was quite powerful for its time: A 333 MHz Pentium II computer running Windows NT 4.0 SP3, using 256 MB of memory. We made sure that the computer was very lightly loaded and that it was never swapping. All tests were run multiple times and the minimum time is reported.

For TLPlan, we used the most recent precompiled version that could be downloaded from http://www.lpaig.uwaterloo.ca/~fbacchus. We used TALplanner version test-68, integrated into VITAL version 2.297. TALplanner is written in Java, and as a runtime system we used the Java Development Kit version 1.2 (http://java.sun.com), the latest version available at the time, together with the Symantec Just In Time Compiler included in the Java Development Kit.

In all cases, TLPlan used the domain definitions and control rules from domains/Blocks/4OpsBlocksWorld.tlp and domains/Logistic/LogisticsWorld.tlp in the TLPlan distribution, respectively, and TALplanner used the corresponding TAL world definition and control rules.

Chapter 7

Concurrency and Resources

Though many planners have been restricted to generating sequential plans, most potential real world applications for planning involve multiple agents or an agent with multiple actuators. For such cases, only generating sequential plans can be a rather severe limitation on plan quality. TALplanner has therefore been extended to generate true concurrent plans. This entails modifying the definition of a plan in order to allow multiple actions to be executed not only in parallel (during identical intervals of time) but also with partially overlapping execution intervals (Section 7.1).

Extending the set of possible plans invalidates certain assumptions that could be made for sequential plans: The planner can no longer assume that the only effects that take place during the execution of an action are those explicitly specified in the definition of that action. This change has wide-ranging consequences for the modeling of planning operators, and the modeling language has to be extended accordingly. For example, there must be a suitable means for restricting concurrency for actions that cannot or should not be executed in parallel, such as driving a truck and loading packages in the same truck in the logistics domain. Several different approaches will be discussed, including an extension to the domain modeling language to allow the use of prevail conditions, an extended form of preconditions not limited to constraining the invocation state of an action (Section 7.2). In addition, concurrent plans are often associated with limited resources such as fuel or cargo space, and though such resources can easily be modeled using plain action effects when plans are sequential, concurrency makes this representation quite inconvenient. An explicit representation of resources is therefore added to the language (Section 7.3). The combination of concurrency, resources and cycle checking is discussed in Section 7.4.

Changing the definition of a plan and introducing new elements into the domain description language also necessitates a number of modifications to the planning algorithm (Section 7.5).

Benchmark tests for the concurrent planner are deferred to Chapter 9, where the results for the Third International Planning Competition (IPC-2002) were generated using the concurrent version of TALplanner.

7.1 Concurrent TALplanner

As with sequential plans, concurrent plans consist of TAL action occurrences of the form $[\tau, \tau']\ o$. Because each action occurrence contains complete timing information, no additional structure is required to capture temporal relations between actions, and a concurrent plan could therefore be represented as an unordered set of action occurrences. In this thesis, though, concurrent plans will usually be viewed as *sequences* of action occurrences, where plans are always extended one action occurrence at a time by the concurrent TALplanner search procedure.

Compared to the definition of sequential plans (Definition 6.2.13 on page 159), the constraint on the time where the action is executed is relaxed: A new action occurrence added to a plan must not start before the start of any existing action occurrence in the current plan, and in order to avoid "gaps" where no action is being executed, it must not start after the end of all existing action occurrence.

Definition 7.1.1 (Concurrent Plan)
A *concurrent plan* for a goal narrative $\mathcal{N}$ is a tuple of ground fluent-free action occurrences with the following constraints. First, the empty tuple is a concurrent plan for $\mathcal{N}$. Second, given a concurrent plan $p = \langle [\tau_1, \tau_1']\ o_1(\bar{c}_1), \ldots, [\tau_n, \tau_n']\ o_n(\bar{c}_n) \rangle$ for $\mathcal{N}$, its successors are exactly those sequences adding one new action occurrence $[\tau_{n+1}, \tau_{n+1}']\ o_{n+1}(\bar{c}_{n+1})$ satisfying the following constraints:

1. Let $\mathcal{N}' = \mathcal{N} \cup \{[\tau_1, \tau_1']\ o_1(\bar{c}_1), \ldots, [\tau_n, \tau_n']\ o_n(\bar{c}_n)\}$ be the original goal narrative $\mathcal{N}$ combined with the existing plan. Then, the new action $o_{n+1}(\bar{c}_{n+1})$ must be applicable over the interval $[\tau_{n+1}, \tau_{n+1}']$ in $\mathcal{N}'$. This implies that its preconditions are satisfied, that its effects are not internally inconsistent and do not contradict the effects of the operator instances already present in the sequence, and that the duration $\tau_{n+1}' - \tau_{n+1}$ is consistent with the duration given in the operator specification.

2. The first action starts at time 0: $\tau_1 = 0$.

3. The new action should not be invoked before any of the actions already existing in the sequence. Therefore, it is required that $\tau_n \leq \tau_{n+1}$. This guarantees that all states up to and including τ_n are *fixed* and will never be modified in any successor of p, which ensures that preconditions of existing actions will never be falsified by new actions being added to the plan.

4. An upper bound will be placed on the invocation timepoint τ_{n+1}. As in Definition 6.4.6, let t_{max} be the maximum of all ending timepoints τ'_i of all actions in p. The states from τ_n up to t_{max} may all be different, but since nothing can change after t_{max}, successors with $\tau_{n+1} > t_{\mathsf{max}}$ are not considered. Thus, it must be the case that $\tau_{n+1} \leq t_{\mathsf{max}}$ (note that it is possible that $t_{\mathsf{max}} > \tau'_n$, as in the operator sequence $\langle [0,7]\ \mathsf{o}_1; [0,3]\ \mathsf{o}_2 \rangle$).

5. For successors where $\tau_{n+1} = \tau_n$ (that is, where the new action has the same invocation timepoint as an existing action), the search tree could contain redundant pairs of plans such as $\langle [0,3]\ \mathsf{o}_1; [0,3]\ \mathsf{o}_2 \rangle$ and $\langle [0,3]\ \mathsf{o}_2; [0,3]\ \mathsf{o}_1 \rangle$. To avoid searching redundant plans, the existence of a total order $\succ$ on operator instances will be assumed, and if $\tau_{n+1} = \tau_n$, it must be the case that $\mathsf{o}_{n+1} \succ \mathsf{o}_n$.
■

As in the definition of sequential plans, this definition induces a possibly infinite search tree which can be traversed using standard search strategies such as breadth first or depth first search. When searching the tree, preference is given to actions invoked at earlier timepoints. In other words, children to any given search node are ordered in such a way that TALplanner will attempt to add as many applicable actions as possible at any given timepoint before stepping to the next timepoint. A partial example search tree for a concurrent version of the gripper domain can be seen in Figure 7.1, where the asterisks indicate undesirable nodes where additional constraints are required to ensure the planner does not pick up two objects in the same gripper or the same object in two grippers (Section 7.2). The corresponding modifications to the search procedures in the TALplanner algorithm are obvious and will not be shown explicitly.

Definition 7.1.2 (Concurrent Solution)
A *concurrent solution* for a goal narrative $\mathcal{N}$ is a concurrent plan p for $\mathcal{N}$ such that $\mathit{Trans}^{+}(\mathcal{N} \cup p) \models \mathit{Trans}(\mathcal{N}_{\mathsf{goal}} \wedge \mathcal{N}_{\mathsf{control}})$. ■

7.1.1 A Concurrent Logistics Domain

In the remainder of this chapter, concepts related to concurrency in TALplanner will be discussed using a concurrent version of the logistics domain.

In a domain such as logistics, it is unreasonable to expect all actions to have the same duration. Therefore, to create efficient plans it is not sufficient to plan actions concurrently, but the planner must also be able to plan a sequence of several "short" actions, like loading, driving and unloading a truck, in parallel with a "long" action, like flying an airplane between distant cities. Therefore, the logistics domain is also extended to make use of actions with variable duration.

The set of value domains and features used in Example 6.2.1 on page 143 is extended as follows. The new integer-valued fluent $\mathsf{dist}(\mathtt{loc}, \mathtt{loc})$ corresponds to the distance between two locations, and the drive and fly actions are modified to

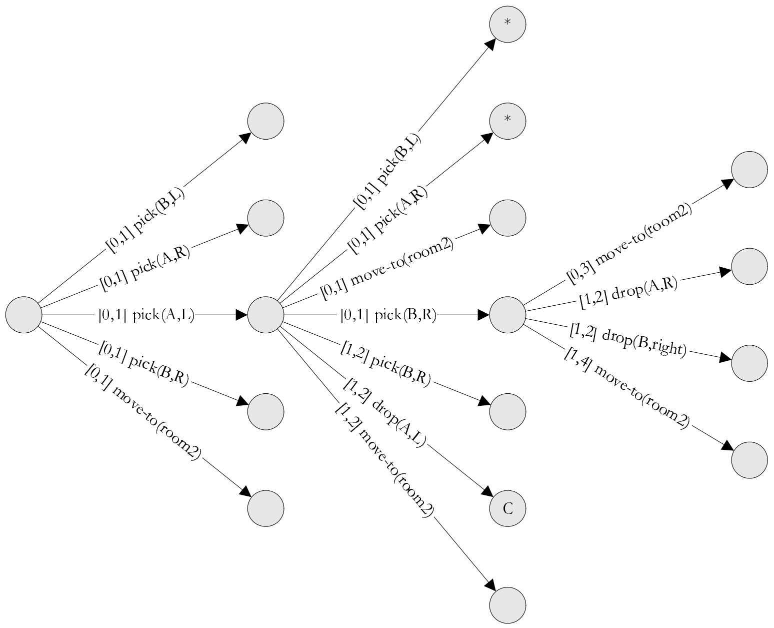


Figure 7.1: Partial Search Space for Concurrent Gripper Domain

take distances into account. The maketime() conversion function converts integers to timepoints as explained in Section 6.2.7 on page 151. The `integer` domain is declared using a special syntax that automatically generates values between the given lower and upper bounds, avoiding the need to explicitly specify each integer value while still remaining within the bounds of TAL which currently requires finite value domains. The modified version of the logistics domain will be used as a basis for presenting concepts related to the concurrent version of TALplanner.

```
domain   integer :integer :lb 0 :ub 10000
domain   loc, thing
domain   airport, city :parent loc
domain   obj, vehicle :parent thing
domain   truck, plane :parent vehicle
feature  at(thing,loc) :domain boolean
feature  in(object,vehicle) :domain boolean
feature  city-of(loc) :domain city
feature  dist(loc, loc) :domain integer
```

The load and unload operators from Example 6.2.10 remain unchanged. The time taken to drive between two locations is chosen to be one half of the distance (in our arbitrary units of distance and time), while the time taken to fly between two airports is chosen to be one fifth of the distance.

```
operator drive(truck, loc_1, loc_2) :at s
 :duration maketime(value(s, dist(loc_1, loc_2))/2)
 :precond [s] at(truck, loc_1) ∧ city-of(loc_1) ≙ city-of(loc_2) ∧ loc_1 ≠ loc_2
 :effects  [s+1] at(truck, loc_1) := false,
           [s+maketime(value(s, dist(loc_1, loc_2))/2)] at(truck, loc_2) := true

operator fly(plane, airport_1, airport_2) :at s
 :duration maketime(value(s, dist(airport_1, airport_2))/5)
 :precond [s] at(plane, airport_1) ∧ airport_1 ≠ airport_2
 :effects  [s+1] at(plane, airport_1) := false,
           [s+maketime(value(s, dist(airport_1, airport_2))/5)] at(plane, airport_2) := true
```

Recall that the minimum allowed duration of an action is one unit of time. Therefore, the minimum valid distance between two locations will in this case be 2 and the minimum valid distance between two airports will be 5, because smaller values will result in a truncated integer duration of 0 for the drive or fly actions. Violations of this rule will be caught by TALplanner during planning.

7.2 Preventing Interference in Concurrent Plans

Any planner supporting concurrent actions must also take into account the fact that certain actions may interfere with each other when executed in parallel, and the domain modeling language used by such a planner must be expressive enough to model the conditions under which concurrency is or is not allowed.

The current formalization of the logistics domain is insufficient in this regard. Given this formalization, the planner could generate a plan which loads packages into a truck while concurrently driving that truck to another location. As long as the load and drive actions begin at exactly the same time, there is no apparent conflict, since load only requires the truck to be at a certain location in the invocation state, not throughout the execution of the action. The planner could also generate a plan where a truck drives to more than one destination concurrently, as long as all concurrent drive actions begin at exactly the same time. Immediately after the actions are invoked, the truck disappears from its original location, exactly as intended ($[s+1]$ $\mathsf{at}(truck, loc_1) :=$ **false**). The first time one of the drive actions finishes executing, the truck ends up at the corresponding destination, again exactly as intended. For each remaining action, the truck will be "cloned", appearing at one additional location ($[s + \mathsf{maketime}(value(s, \mathsf{dist}(loc_1, loc_2))/2)]$ $\mathsf{at}(truck, loc_2) :=$ **true**).

Though it may seem somewhat surprising, this is in fact not a flaw in the planning algorithm itself. The plans being generated are correct according to the semantics of concurrent TAL narratives – we have simply failed to provide sufficiently strong executability conditions, so our models of the load and drive operators are incorrect for concurrent plans. But is this only a failure in our domain description, or is the current domain description language in fact not expressive enough to model these operators correctly? As we will see below, both points could be argued. Many constraints on concurrency can be modeled in the current language, but not necessarily in a convenient manner: It may be necessary to "misuse" some language constructions to achieve the intended semantics in a roundabout way. We will now explore how existing constructions can be used and misused, and introduce a new extension to the current language in order to model certain kinds of interference in a more elegant manner. In Section 7.3.5, we will also show how interference can be prevented using resource constraints.

7.2.1 Preventing Interference Using Preconditions

Some planners and modeling languages have chosen to implicitly interpret preconditions as conditions that must hold throughout the execution of an action rather than only in the state where the action is invoked. This would prevent concurrent invocations of load and drive for the same truck, since executing a load action would automatically require the truck to remain at its initial location until the end of the action.

In our opinion this interpretation of preconditions is too inflexible, making it impossible to model actions for which a condition only needs to hold during part of the execution interval. The fact that fluents explicitly modified by an action must be exempt from the implicit requirement to remain constant throughout its execution also detracts from the elegance of the solution. Additional complications in determining exactly which fluents should be allowed to be modified arise when quantified and non-conjunctive preconditions are allowed and when temporally extended actions are used. For example, given the precondition $\alpha \vee \beta$, must all fluents involved in these formulas remain unmodified or should it be sufficient that $\alpha \vee \beta$ holds throughout the interval even though individual fluents may change? This complexity would make the precise meaning of a precondition unnecessarily difficult to foresee, whereas the current semantics of TALplanner preconditions is comparatively simple.

But if preconditions only constrain the invocation state of an action, and action effects cannot affect this state, then no straight-forward scheme to model mutual exclusion between two actions using preconditions will work. Clearly, whether or not a precondition of an action invoked at time t is satisfied can only depend on actions invoked at times strictly before t (so that their effects may affect the new invocation state at t), not on actions invoked exactly at t. In the logistics example, the precondition of a drive action can never depend on whether a load action has already been added at the same timepoint, and vice versa.

In this particular example, this restriction could be worked around by introducing a new feature whose value determines whether the planner is allowed to drive or load any given truck at a given timepoint, together with a "pseudo-operator" that can be inserted to change the value of this feature. The load and drive actions would be conditioned on the value of the new feature, approximately as follows:

```
domain      task :elements { shouldload,shoulddrive }
feature     task-of(vehicle) :domain task
operator    choose-task(vehicle, task) :at s
  :precond  true
  :effects  [s+1] task-of(vehicle) := task
operator    load(obj, vehicle, loc) :at s
  :precond  [s] task-of(vehicle) ≙ shouldload ∧ ...
```

Since the task of a vehicle could never be both **shouldload** and **shoulddrive** at the same time, we could never try to load and drive the vehicle concurrently, and therefore the problem is in a sense solved.

Though this scheme would work, it is far from straight-forward. Because it involves introducing "false" actions that have no correspondence in reality and will have to be filtered out from the solution plan, we view it as a misuse of the constructs provided by the current language. Given a choice, we would instead prefer to use constructs explicitly defined for supporting the modeling of concurrent actions. We therefore immediately proceed to the next potential approach: Using prevail conditions.

7.2.2 Preventing Interference Using Prevail Conditions

If using preconditions to induce a set of implicit constraints on the development of the world during the entire execution of an action would lead to an overly complex semantics, the natural solution would be to introduce a new construction permitting such constraints to be modeled explicitly in the description of a planning domain. Constraints that should hold throughout the execution of an action are sometimes called *prevail conditions* (Sandewall & Rönnquist, 1986; Bäckström & Klein, 1991). We will generalize this terminology somewhat, though, permitting prevail conditions to refer to arbitrary timepoints or intervals during the execution of an action. The semantics of a prevail condition is simple: The prevail condition of each action in a plan must be entailed by the plan, with no exceptions for fluents modified by the action to which the condition belongs.

The syntax of an *operator specification*, previously defined in Definition 6.2.9 on page 152, is extended to support prevail conditions. The definition of a context specification remains unchanged.

Definition 7.2.1 (Operator Specification)
An *operator specification* in $\mathcal{L}(\mathrm{ND})^*$ is a labeled statement having the following form, where o is an operator name, v_1 through v_n are distinct value variables serving as formal parameters, the invocation timepoint s is a temporal variable, the duration specification τ is a temporal term, the precondition ϕ is a single timepoint formula for s, the prevail condition ψ is a static formula, and c_1 through c_m are context specifications for the invocation timepoint s.

- **operator** o$(v_1, \ldots, v_n)$:at s :duration τ :precond ϕ :prevail ψ :context c_1 … :context c_m

Omitting the precondition specification (:precond ϕ) is equivalent to specifying :precond `true`. Omitting the prevail condition specification (:prevail ψ) is equivalent to specifying :prevail `true`. Omitting the duration specification is equivalent to specifying :duration 1. For actions with only one context specification, the :context keyword can be omitted. ■

The translation function *Trans* is modified as follows to account for the addition of prevail conditions.

- $\mathit{Trans}_{\mathsf{op}}$(operator o$(v_1, \ldots, v_n)$:at s :duration τ :precond ϕ :prevail ψ :context c_1 … :context c_m) =
  $\forall s, s', v_1, \ldots, v_n.$
  $\mathit{Occurs}(s, s', \mathsf{o}(v_1, \ldots, v_n)) \rightarrow s' = s + \tau \wedge \mathit{Trans}(\phi \wedge \psi \rightarrow \bigwedge_{i=1}^{m} \mathit{Trans}_{\mathsf{con}}(s, c_i)).$

In the context of sequential planning, prevail conditions would only have been able to constrain the effects of the operator for which the condition was declared. For concurrent planning, on the other hand, prevail conditions also constrain the effects of all other actions executing within the same interval of time. We can immediately see that this can be used to model the intended executability conditions for the load operator in the logistics example: The vehicle into which a package is being loaded must remain at its location throughout the execution of the action.

Example 7.2.1 (Preventing Interference using Prevail Conditions)

operator	load(*obj*, *vehicle*, *loc*) :at *s*
:precond	[*s*] at(*obj*, *loc*) ∧ at(*vehicle*, *loc*)
:prevail	[*s*,*s*+1] at(*vehicle*, *loc*)
:effects	[*s*+1] at(*obj*, *loc*) := **false**, [*s*+1] in(*obj*, *vehicle*) := **true**

■

Testing Prevail Conditions

Preconditions are tested immediately before a new action is added to a plan. At this time, the planner already has complete knowledge about the single state in which the precondition is tested. Due to the structure of the search space used by TALplanner and the requirement that no action can have effects in or before its own invocation state, we know this state will remain unaltered by the action whose precondition is being tested as well as by any other action that might be added to the plan during the search for a solution.

Prevail conditions are fundamentally different in this respect. They refer to states that are conceptually in the future, and the planner's knowledge about these states is generally incomplete. But this does not necessarily mean these states are completely unknown. The concurrent planner may already have added one or more actions that have effects in what is currently the future, and the fluent values assigned by these effects are definite: They cannot be altered by any means other than backtracking.

Since TALplanner correctly models this incomplete knowledge (by occluding fluents in the "future" and thereby releasing them from the inertia assumption), there is no need to wait until a state is in the "past" before testing a prevail condition in that state. On the contrary, prevail conditions can be tested at any time, with three possible outcomes:

- The prevail condition is true (entailed by the current plan). In this case, it will necessarily remain true no matter which actions are added to the plan in the future, as long as the TALplanner search procedure is followed. The condition does not have to be tested again.
- The prevail condition is false (its negation is entailed by the current plan). In this case, it will necessarily remain false no matter which actions are added to the plan in the future, as long as the TALplanner search procedure is followed. The planner must backtrack.
- The status of the prevail condition cannot be determined (neither the condition nor its negation is entailed by the current plan). In this case, the condition must be queued, to be tested again when additional information is available.

The fact that prevail conditions can be tested in incompletely specified states is crucial to the performance of the planner for many domains. If the planner had to wait until a state was completely known before testing a prevail condition, there would potentially be an exponential explosion in the number of plans to explore.

Example 7.2.2 (Avoiding Exponential Backtracking)
Consider the concurrent version of the logistics domain. Assume that the planner has already added the action $[0,1]$ load(**package-1**, **truck-1**, **city1-1**) to the plan and is currently considering instances of the drive operator.

The first applicable instance of drive might be $[0,4]$ drive(**truck-1**, **city1-1**, **city1-2**). Because there may be additional actions applicable at time 0 that could have effects at time 1, the information available about the state at time 1 is still incomplete. Nevertheless, once the drive action is added, sufficient information is already available to determine that the prevail condition of the preceding load action is violated, and the planner will immediately backtrack.

Had complete information about a state been required, no violation would have been discovered immediately. Instead, the planner would have continued adding actions invoked at time 0, until no more such actions could be found. At this time, the planner would step to time 1 and finally detect the violation. Backtracking would remove one action, after which the planner would once more step to time 1 and detect the same violation. This process would continue until all possible subsets of actions applied after $[0,4]$ drive(**truck-1**, **city1-1**, **city1-2**) had been tested, at which time this action would finally be removed. ■

The use of prevail conditions neatly solved the problem of ensuring that trucks do not leave while packages are being loaded. On the other hand, ensuring that trucks do not drive to two different destinations concurrently may be done simply by making the domain model slightly less abstract.

7.2.3 Preventing Interference Using Action Effects

If the effects of a new action added to a plan contradict the effects of an existing action, the resulting narrative is inconsistent and TALplanner is guaranteed to backtrack. In some domains, a sufficient degree of mutual exclusion can therefore follow automatically from action effects, as long as the domain model is sufficiently detailed.

For example, the planner can be prevented from allowing a truck to drive to multiple locations at the same time if two new fluents are added: moving(`vehicle`), a boolean fluent which holds when a vehicle is moving, and destination(`vehicle`) : `loc`, which represents the latest assigned destination of the vehicle. An interval effect is used to ensure that the destination of a vehicle remains the same throughout the execution of a drive action. Two concurrent drive actions for the same truck will attempt to assign different values to the destination, forcing the planner to backtrack.

```
feature     moving(vehicle, loc) :domain boolean
feature     destination(vehicle) :domain loc

operator    drive(truck, loc1, loc2) :at s
  :duration maketime(value(s, dist(loc1, loc2))/2)
  :precond  [s] at(truck, loc1) ∧ city-of(loc1) ≙ city-of(loc2) ∧ loc1 ≠ loc2
  :effects  [s+1] at(truck, loc1) := false,
            [s+maketime(value(s, dist(loc1, loc2))/2)] at(truck, loc2) := true,
            [s+1,s+maketime(value(s, dist(loc1, loc2))/2-1)] moving(truck) := true,
            [s+maketime(value(s, dist(loc1, loc2))/2)] moving(truck) := false,
            [s+1,s+maketime(value(s, dist(loc1, loc2))/2)] destination(truck) := loc2
```

One point deserves further clarification. Given that the distance between two locations is sufficiently small, the duration of the operator will be 1 and the interval effect for moving(*truck*) will take place at $[s+1, s+0]$. This is a valid specification of a null effect – the fluent is affected at all timepoints within the interval, that is, at no timepoints at all. Though this may appear somewhat strange, it achieves the intended purpose of ensuring that moving(*truck*) is forced to be true in the inner part of the execution interval (though the inner part in this case happens to be empty), while the second effect for moving(*truck*) forces the fluent to be false at the end point of the execution interval.

7.2.4 Preventing Interference Misusing Action Effects

Having successfully modeled mutual exclusion between multiple drive operators using action effects, let us now go on to see whether we could also have used the same technique to ensure that trucks do not leave while packages are being loaded.

As it turns out doing this is not too difficult: The load operator can be modified by adding an effect causing the vehicle to be at its original location throughout the execution of the operator. Attempting to drive the same vehicle to another location will then cause the vehicle *not* to be at its original location, contradicting the effects of the load operator. This will be true regardless of the order in which the two operators are added to a plan, ensuring mutual exclusion.

```
operator load(obj, vehicle, loc) :at s
 :precond [s] at(obj, loc) ∧ at(vehicle, loc)
 :effects [s+1] at(obj, loc) := false, [s+1] in(obj, vehicle) := true
          [s+1] at(vehicle, loc) := true
```

Although this in some sense achieves our intentions, it should more properly be regarded as a misuse of action effects. Loading a package into a vehicle does not physically *cause* the vehicle to remain where it is; a truck could easily start driving while a package is being loaded. Though the plans generated by this variation of the logistics domain may be correct, our model of the operators in this domain no longer reflects reality.

This is naturally not a proof of non-existence: It may still be possible to find other aspects of the domain that can be modeled in order to prevent interference between drive and load using action effects. The example is merely intended as a concrete demonstration of a case where this modeling technique is misused.

7.3 Modeling Limited Resources

Many planning domains involve the use of limited resources which can be consumed, produced, borrowed and returned, or used in various other ways. For example, vehicles such as trucks and airplanes can have limited carrying capacities and a limited amount of fuel available.

For sequential planning, such properties can usually be modeled quite easily in TALplanner by using plain action effects updating the values of non-boolean fluents. For example, if loading a package into a vehicle requires one unit of space, the amount of available space could be decreased as follows:

```
operator load(obj, vehicle, loc) :at s
:precond [s] at(obj, loc) ∧ at(vehicle, loc)
:effects [s+1] space(vehicle) := value(s, space(vehicle)) − 1, ...
```

With concurrent planning this is clearly not sufficient, since multiple parallel invocations of load would not take cumulative concurrent effects into account: Given

n units of space available, each parallel invocation of load would assign the same new value $n-1$, for a total consumption of one unit of space regardless of the total number of packages loaded. Even for sequential planning, adding explicit built-in support for resources often facilitates the task of writing domain definitions. Therefore, we introduce explicit support for resources in TALplanner.

7.3.1 Declaring and Using Resources

Like all entities present in a planning domain, resources must be declared before being used. Resource declaration statements are similar to fluent declaration statements and have the following form:

resource res($\overline{x}$) :domain *domain*

For example, in a problem domain where vehicles have limited space, a space resource can be declared as follows:

resource space(vehicle) :domain integer

A resource sort is added to $\mathcal{L}(\text{ND})^*$. Resource terms are formed from resource symbols in the same manner as fluent terms are formed from feature symbols.

Definition 7.3.1 (Resource Sorts)
There are a number of sorts for features $\mathcal{R}_i$, each one associated with a value domain $dom(\mathcal{R}_i) = \mathcal{V}_j$ for some j. The sort $\mathcal{R}$ is assumed to be a supersort of all resource sorts. ■

Definition 7.3.2 (Resource Term)
A *resource term*, often denoted by r, is a resource expression $\text{res}(\omega_1, \ldots, \omega_n)$ where $\text{r} : \mathcal{V}_{k_1} \times \ldots \times \mathcal{V}_{k_n} \rightarrow \mathcal{R}_i$ is a resource symbol and each ω_j is a value term of sort $\mathcal{V}_{k_j}$. ■

Definition 7.3.3 (Single Timepoint Resource Term)
A *single timepoint resource term* for the timepoint τ is a resource term where all occurrences of the *value* function are of the form $value(\tau, f)$ (where f is a fluent term). ■

The definition of context specifications from Section 6.2.7 on page 151 is extended in order to provide a structured way of declaring the resource usage of an operator.

Definition 7.3.4 (Context Specification)
A *context specification* for the invocation timepoint s has the following form, where v_1 through v_n are distinct value variables, ϕ is a single timepoint formula for s, r_1 through r_p are resource effect expressions for s, and e_1 through e_m are effect expressions for s:

- :forall $v_1, \ldots, v_n$:condition ϕ :resources $r_1, \ldots, r_p$:effects $e_1, \ldots, e_m$

If quantification is not required, the quantifier section (:forall $v_1, \ldots, v_n$) can be omitted. If no context condition is required, the condition section (:condition ϕ) can be omitted; this is equivalent to specifying :condition `true`. If no resource effects are required, the resource effect section can be omitted. ■

While an ordinary effect expression provides a definite new value for a fluent, resource effects must be cumulative, with the planner taking all concurrent effects on each resource into account in order to calculate the amount available at any point in time. We therefore introduce a new form of resource effect expression, which specifies the resource which is affected by the action, the time at which it is affected, and *how* it is affected.

Unlike some planners, TALplanner only provides one type of resource, but allows several different types of resource effects. Resources can be *produced* and *consumed*, which adds or removes a given amount from what is currently available. It is also possible to *assign* an absolute value to a resource, stating the exact amount that will be available. A resource can be *borrowed*, which means that the action reserves the resource at a timepoint or throughout an interval of time, thereafter automatically returning it to a common pool of available resources. Finally, resources can be *borrowed non-exclusively*, allowing a set of cooperating concurrent actions to reserve the same set of resources. The exact semantics of these different types of resource effects will be defined later in this section.

Definition 7.3.5 (Resource Effect Expression)
A *resource effect expression* for the invocation timepoint s has one of the following forms, where τ and τ' are temporal terms, r is a single timepoint resource term for s and ω is a single timepoint value term for s of a sort corresponding to the sort of f:

- $[s + \tau]$:produce r :amount ω
- $[s + \tau]$:consume r :amount ω
- $[s + \tau]$:borrow r :amount ω
- $[s + \tau]$:borrow-nonex r :amount ω
- $[s + \tau, s + \tau']$:borrow r :amount ω
- $[s + \tau, s + \tau']$:borrow-nonex r :amount ω
- $[s + \tau]$:assign r :amount ω ■

Example 7.3.1 (Space in the Logistics Domain)
In the following example, loading a package into truck always consumes one unit of space.

operator load(*obj*, *vehicle*, *loc*) :at s
  :precond [s] at(*obj*, *loc*) $\wedge$ at(*vehicle*, *loc*)
  :effects [s+1] at(*obj*, *loc*) := **false**, [s+1] in(*obj*, *vehicle*) := **true**
  :resources[s+1] :consume space(*vehicle*) :amount 1 ■

7.3.2 Querying Resources

Each resource has a number of different aspects modeled as fluent macros.

First, in any state, there is an *initial* amount of resources – the amount that would be available if no resource effects took place in that state. Given a resource res, this is modeled as a fluent macro init(res). The initial amount can be queried in any state, and must be provided through observation statements for the initial state:

$$\textbf{init} \quad \forall truck\,[\,[0]\ \mathsf{init}(\mathsf{fuel}(truck)) \ \hat{=}\ 30\,]$$

Second, in any given state, certain amounts of each resource res may have been produced, consumed, borrowed non-exclusively, and borrowed exclusively. These amounts may arise from cumulative effects of a number of concurrent operators, and can be referred to using the fluent macros produced(res), consumed(res), borrowed-nonex(res), and borrowed(res), respectively.

Third, it is often useful to be able to refer to the amount of resources actually available for consumption in any given state. This amount can be referred to as available(res). If res has been assigned a new value at the current timepoint, then available(res) is equal to this new value. Otherwise, it is equivalent to init(res) − consumed(res) − borrowed-nonex(res) − borrowed(res) (recall that resources produced in a state are not available for consumption in the same state).

Similarly, one can refer to the amount of resources transferred to the next state as transferred(res). If res has been assigned a new value at the current timepoint, then transferred(res) is equal to this new value. Otherwise, it is equivalent to init(res) + produced(res) − consumed(res).

Providing a completely TAL-based semantics for these macros is not possible, since it requires summing over an unbounded number of actions in the current plan and over all those instances of quantified conditional effects that actually take place and affect the particular resource in which we are interested. Support for such sums is expected to be available in a future version of the planner, based on a new, modified $\mathcal{L}(\mathrm{FL})$ logic. For the moment, each fluent macro is provided with a value through semantic attachment. Resource effects are also handled through semantic attachment, and are ignored in the $\mathcal{L}(\mathrm{FL})$ translation of an operator definition.

7.3.3 Built-In and Complex Resource Constraints

For each resource res, the minimum and maximum amounts allowed can be specified using the macros minimum(res) and maximum(res). These values must be specified in the initial state. For example, the fact that it is possible to have between 0 and 60 units of fuel can be specified as follows:

$$\textbf{init} \quad \forall truck\,[\,[0]\ \mathsf{minimum}(\mathsf{fuel}(truck)) \ \hat{=}\ 0 \wedge \mathsf{maximum}(\mathsf{fuel}(truck)) \ \hat{=}\ 60\,]$$

In states where a resource has been explicitly assigned a new value, it is sufficient to ensure that the new value is within the allowed range. When no assignment has taken place, however, there may be a number of concurrent resource effects affecting the same resource. Here, TALplanner uses a semantics where resources produced in one state are available for consumption in the next state. For the minimum constraint, this entails the assumption that all consumption in any given state might take place before any production, leading to the constraint that $\mathsf{init}(\mathsf{res}) - \mathsf{consumed}(\mathsf{res}) - \mathsf{borrowed\text{-}nonex}(\mathsf{res}) - \mathsf{borrowed}(\mathsf{res}) \geq \mathsf{minimum}(\mathsf{res})$ in all states, or, equivalently, that $\mathsf{available}(\mathsf{res}) \geq \mathsf{minimum}(\mathsf{res})$ in all states. Similarly, to ensure that the amount of resources never exceeds the specified maximum, the planner requires that $\mathsf{init}(\mathsf{res}) + \mathsf{produced}(\mathsf{res}) \leq \mathsf{maximum}(\mathsf{res})$ at all timepoints.

Note that since resource macros can be used in control rules, one is not limited to these simple minimum/maximum constraints on resources. For example, it is easy to state that no more than 5.5 units of fuel can be consumed at any given timepoint, that it is impossible to produce and consume units of the same resource at the same time, that equal amounts of two resources must always be available, or even a complex constraint such that whenever some condition ϕ holds, a certain resource may not be consumed during the following 6 timepoints:

control $\forall t.\ [t]\ \mathsf{consumed(fuel)} \leq 5.5$
control $\forall t.\ [t]\ \mathsf{consumed(res)} \hat{=} 0 \vee \mathsf{produced(res)} \hat{=} 0$
control $\forall t.\ [t]\ \mathsf{available(res)} \hat{=} \mathsf{available(res')}$
control $\forall t.\ [t]\ \phi \rightarrow [t{+}1,t{+}6]\ \mathsf{consumed(res)} \hat{=} 0$

7.3.4 Modeling Symmetric Objects using Resources

In addition to modeling limited availability, resources can also be used to model symmetric objects.

Consider once more the gripper domain. In the case where the robot has more than one gripper, one difficulty for a planner is recognizing that the grippers are functionally identical; if a plan can not be completed using the left gripper to pick up a certain object, using the right gripper instead will not fix the problem either (this was also pointed out in Fox & Long, 1999). This difficulty can be avoided by modeling the grippers as a resource of bounded capacity. Picking up and dropping objects "consumes" and "produces" grippers, respectively. This also facilitates the definition of problem instances: Rather than having to name each gripper, it suffices to specify the number of grippers that are available.

The value domains `obj` for things (including the robot), `ball` for balls (a subtype of `obj`), and `room` for rooms are unchanged from the previous formalization of this domain. The `integer` value domain is added and the `gripper` value domain is no longer used. The fluent loc is unchanged, while the fluent free is replaced with the resource free-gripper and the fluent is-carried-in is replaced with is-carried which does not have a `gripper` argument.

```
domain    obj :elements { ball1, ball2, ball3, robby }
domain    ball :parent obj :elements { ball1, ball2, ball3 }
domain    room :elements { roomA, roomB }
resource  free-gripper :domain integer
feature   loc(obj) :domain room
feature   is-carried(obj) :domain boolean
init      [0] init(free-gripper) ≙ 2 ∧ minimum(free-gripper) ≙ 0 ∧ maximum(free-gripper) ≙ 2
```

Given these definitions, the gripper domain operators shown in Example 6.2.11 on page 158 can be redefined as follows:

```
operator pick(ball) :at s
  :precond [s] loc(ball) ≙ loc(robby) ∧ ¬is-carried(ball)
  :resources [s+1] :consume free-gripper :amount 1
  :effects [s+1] is-carried(ball) := true

operator drop(ball) :at s
  :precond [s] is-carried(ball)
  :resources [s+1] :produce free-gripper :amount 1
  :effects [s+1] is-carried(ball) := false

operator move-to(room) :at s
  :precond [s] loc(robby) ≙̸ room
  :context
    :effects [s+1] loc(robby) := room
  :context :forall ball :condition [s] is-carried(ball)
    :effects [s+1] loc(ball) := room
```

7.3.5 Preventing Interference using Resources

Resources can also be used to model semaphores, which can be used as a complement to prevail conditions and ordinary action effects in order to prevent interference between actions. The following statements introduce a new resource use-of(thing) to be used for this purpose in the logistics domain.

```
resource use-of(thing) :domain integer
init [0] ∀thing [ init(use-of(thing)) ≙ 0 ∧ minimum(use-of(thing)) ≙ 0 ∧
                  maximum(use-of(thing)) ≙ 1 ]
```

The use-of resource ensures that an object or vehicle is never used in conflicting concurrent actions. When packages are loaded into or unloaded from a vehicle, the corresponding use-of resources are borrowed non-exclusively, allowing several loading or unloading actions involving the vehicle to take place concurrently. Actions that move the vehicle borrow the resource exclusively, so that it can never be moved to two different destinations at the same time or moved during loading or unloading.

```
operator load(obj, vehicle, loc) :at s
 :precond [s] at(obj, loc) ∧ at(vehicle, loc)
 :resources  [s+1] :borrow-nonex use-of(vehicle) :amount 1,
             [s+1] :borrow use-of(obj) :amount 1
 :effects [s+1] at(obj, loc) := false, [s+1] in(obj, vehicle) := true

operator drive(truck, loc1, loc2) :at s
 :duration maketime(value(s, dist(loc1, loc2))/2)
 :precond [s] at(truck, loc1) ∧ city-of(loc1) ≙ city-of(loc2) ∧ loc1 ≠ loc2
 :resources  [s+1,s+maketime(value(s, dist(loc1, loc2))/2)] :borrow use-of(truck) :amount 1,
 :effects  [s+1] at(truck, loc1) := false,
           [s+maketime(value(s, dist(loc1, loc2))/2)] at(truck, loc2) := true
```

7.4 Concurrency, Resources and Cycle Checking

Cycle checking prunes a plan candidate p if the final state generated by p is identical to the final state generated by a proper prefix of p. Thus, in order to perform cycle checking, all final states of proper prefixes of the current plan candidate must be available for comparison.

For sequential plans, all such states are already available in the state sequence generated by the current plan candidate. For example, the logistics plan candidate $\langle\rangle$ (the empty plan) generates a final state at time 0. Adding the action occurrence $[0, 4]$ drive(**truck-1**, **city1-1**, **city1-2**) generates a new final state at time 4, but the final state of the parent plan remains unaltered and can still be retrieved from the state sequence for the extended plan. Adding yet another action occurrence $[4, 5]$ load(**package-1**, **truck-1**, **city1-2**) generates a new final state at time 5 but preserves the states at times 0 and 4, and so on.

For concurrent plans, though, adding a new action may change the contents of what was previously a final state. Consider again the logistics plan candidate $\langle[0, 4]$ drive(**truck-1**, **city1-1**, **city1-2**)$\rangle$, which generates a final state at time 4. If a new concurrent action $\langle[0, 4]$ drive(**truck-2**, **city2-1**, **city2-2**)$\rangle$ is added to this plan, the state at time 4 will change due to new effects being applied to this state. For this reason, the concurrent planning algorithm must be altered to keep an explicit list of the final states of all proper prefixes of the current plan. Memory usage for this list is generally rather low, given the extensive structure sharing used by the TALplanner implementation. Efficiency is improved by defining an arbitrary total order on states, storing the list in sorted order, and using binary search to determine whether a state is already present in the list.

The concept of cycle checking can be generalized by pruning a plan candidate if the final state generated by the this plan is, by some definition, *worse* than the final state generated by a proper prefix of the same plan. In most cases it may be conceptually or computationally difficult to determine whether one state is worse than another, and therefore this concept is not implemented in most forward-chaining planners. This is not necessarily the case in the presence of explicitly modeled resources, however. Resources are often associated with a preference for greater, or occasionally lesser, amounts of the resource. For example, in most domains where driving consumes fuel, a state where more fuel is available is always better, all else being equal. A plan where one drives from some location **A** to **B** and immediately back to **A** should therefore be pruned, despite the fact that it does not generate a true state cycle: Less fuel is available after driving, so the new state is not identical to, but worse than, the original state.

TALplanner allows each resource to be associated with a preference: more, less, or none. The syntax for resource declarations is extended accordingly:

resource res($\overline{x}$) :domain *domain* [:preference (:more | :less | :none)]

This induces a partial order on states, better-or-equal, used in the concurrent version of TALplanner: better-or-equal(s, s') holds iff (1) s and s' are equal wrt. ordinary fluents and resources with preference none, (2) for every resource with preference more, there is at least as much available in s as in s', and (3) for every resource with preference less, there is at least as much available in s' as in s.

Binary search can still be applied to the fluent portion of a state when searching for a visited state which is better than or equal to the final state of the current plan, preserving most of the efficiency of the standard cycle checking procedure.

7.5 Concurrent TALplanner

Though the concurrent TALplanner algorithm could theoretically allow the use of tense control rules, our efforts have been focused on the use of TAL-based control rules. The concurrent algorithm below is therefore based on the TAL-based TALplanner algorithm from Definition 6.4.8 on page 174, and all indicated differences are relative to that algorithm.

The main differences in this variation of the algorithm relate to the fact that after adding a new action, the planner will only have complete knowledge of the states up to and including the invocation timepoint of that action, rather than the end timepoint. Fluents in the narrative are therefore occluded up to and including time τ_0 rather than time τ. Additional changes to the original algorithm are caused by the addition of prevail conditions, which may also have to be queued in the condition queue for future testing. Because the end timepoints of action occurrences in a concurrent plan are not necessarily monotonically increasing, the maximum of all end timepoints is explicitly stored in each search node (τ_{max}). Finally, some changes are necessitated by the differences between the sequential search tree and the concurrent search tree, as seen in the iteration over possible successor actions.

Definition 7.5.1 (Concurrent TALplanner)
Input: A goal narrative $\mathcal{N}$.
Output: A plan narrative entailing the goal $\mathcal{N}_{\mathsf{goal}}$ and the control formulas $\mathcal{N}_{\mathsf{control}}$.

```
 1   procedure TALplanner-concurrent($\mathcal{N}$)
 2   $\gamma \leftarrow \bigwedge \mathcal{N}_{\mathsf{goal}}$                                   Conjunction of all goal statements
 3   $\langle \mathsf{init}, \mathsf{incr}, \mathsf{final} \rangle \leftarrow$ generate-pruning-constraints($\mathcal{N}_{\mathsf{control}}$)
▷4   node $\leftarrow \langle \mathsf{init}, \emptyset, 0, 0, \langle\rangle \rangle$   ⟨cond. queue, visited states, latest invocation time, $t_{\mathsf{max}}$, plan⟩
 5   Open $\leftarrow \langle \mathsf{node} \rangle$                                   Stack (depth first search)
 6   while Open $\neq \langle\rangle$ do
▷7     $\langle C, S, \tau_0, \tau_{\mathsf{max}}, p \rangle \leftarrow$ pop(Open)                   Current plan candidate
 8     $\mathcal{N}' \leftarrow \mathcal{N} \cup p \cup$ occlude-all-after($\mathcal{N}, \tau_0$)        No knowledge about future
 9     for all constraints $\alpha$ in $C$ do                             Check queued constraints
10       if $Trans^+(\mathcal{N}') \models Trans(\alpha)$ then $C \leftarrow C \setminus \{\alpha\}$    Remove satisfied constraint
11       elsif $Trans^+(\mathcal{N}') \models Trans(\neg\alpha)$ then backtrack           Constraint violated
▷12    $\mathcal{N}'' \leftarrow \mathcal{N} \cup p \cup \{t_{\mathsf{max}} = \tau_{\mathsf{max}}\}$               Narrative with complete knowledge
13     if $Trans^+(\mathcal{N}'') \models$ false then backtrack         Consistency check (def. 6.2.12)
▷14    if $\exists s \in S$.better-or-equal($s$, final state of current plan) then backtrack
15     if $Trans^+(\mathcal{N}'') \models Trans(\gamma \wedge C \wedge \mathsf{final})$ then   Goal + queued + final ctrl satisfied
16       return $\mathcal{N}''$
17     else                                               Not a solution, but check children
►18      $S' \leftarrow S \cup \{$final state of current plan$\}$
▷19      for all successor actions $\mathbf{A} = [\rho, \rho']\ \mathsf{o}^i(\bar{c})$ for $p$ according to Def 7.1.1 do
▷20        $C' \leftarrow C \cup \mathsf{incr}_i[\rho, \bar{c}]$                          Old conditions + incr control
►21        $C' \leftarrow C' \cup \{$prevail condition of $\mathbf{A}\}$               Add prevail condition
▷22        push $\langle C', S', \rho, \max(\tau_{\mathsf{max}}, \rho'), \langle p; \mathbf{A} \rangle \rangle$ onto Open
23   fail                                                                  ■
```

Chapter 8

Domain Analysis Techniques for Domain-Dependent Control

Preliminary benchmark tests for TALplanner were quite satisfying, both compared to fully automated planners and compared to TLPlan. Despite this, it might be possible to make even better use of the knowledge we have about planning domains, taking the performance of a hand-tailored planner to a new level. There are many different avenues to be explored in this area. Perhaps the control formulas used by TLPlan and TALplanner are not the optimal form in which our current domain knowledge can be described. Perhaps the language should be extended to allow *new* types of domain knowledge to be expressed, knowledge that cannot be described in terms of constraints on a single narrative. Some efforts have been made to explore those paths, but our main focus has been on the use of automated domain analysis.

Numerous domain analysis techniques for planning domains already exist in the literature (Nebel, Dimopoulos, & Koehler, 1997; Haslum & Jonsson, 2000; Fox & Long, 1998; Cresswell, Fox, & Long, 2002; Fox & Long, 2000b, 2000a, 2002; Gerevini & Schubert, 1998, 2000; Scholz, 2000; Rintanen, 2000b), and the first task to be performed was therefore an investigation of these techniques to determine whether or not they would be applicable to TALplanner. As it turned out, most existing techniques would require major extensions and modifications, for several reasons.

First, the control rules used by TALplanner are essentially specifications of temporally extended goals. These rules constrain the paths by which the planner is allowed to reach a goal state, but several analysis methods depend on the fact that only the final state is constrained and that the path by which this state is reached is irrelevant. Applying such techniques within the TALplanner framework may render the planner incomplete.

Second, one of the design goals for TALplanner is the ability to plan for domains with large numbers of objects and operator instances. Even if an operator could have billions of instances, this should not be a major problem as long as sufficiently strong control rules can be written to guide the planner towards choosing "good" instances to be applied. For this reason, techniques that rely on generating all ground instances of operators or predicates are less likely to be useful in conjunction with TALplanner. This includes techniques such as RIFO (Nebel et al., 1997) and the methods for planning with reduced operator sets developed by Haslum and Jonsson (2000).

Finally, another important design goal is that of permitting the use of more complex types of operators, including operators with extended duration and (eventually) non-deterministic effects, as well as the use of resources and concurrency. Any techniques depending on the use of single-step operators would require extensions in order to be used in TALplanner.

Given these restrictions, it may be better to start not by examining analysis techniques for those aspects of a planning domain that are common between TALplanner and other planners but by considering techniques that would be applicable to the main difference: Control formulas. The use of control formulas and incremental pruning constraints can significantly reduce the number of search nodes the planner must expand and investigate, which in turn decreases the time spent applying the effects of an action to generate a single new node or testing whether the goal is satisfied by the current plan candidate, but also increases the amount of time spent determining whether or not a node actually satisfies all incremental pruning constraints. In fact, benchmarks show that evaluating incremental pruning constraints often accounts for more than 99% of the time used by TALplanner. This makes formula evaluation performance paramount for the overall performance of the planner, and any techniques that can reduce the amount of time required to verify whether a node should indeed be accepted could become valuable weapons in the hunt for efficiency.

There are several potential approaches that might decrease the amount of time used for evaluating incremental pruning constraints. One of these approaches would involve retaining the same pruning constraints but optimizing the formula evaluator. This has been done in several stages, but the optimizations are mainly related to implementation details and are not interesting in themselves. Another approach would be to use some form of domain analysis to alter the incremental pruning constraints, perhaps finding simpler constraints that would yield the same results in terms of pruning. That is the approach taken in this chapter, based on an ICAPS-2002 article (Kvarnström, 2002).

Borrowing a term from compiler technology, we say that TALplanner contains an *optimizer* for formulas and terms. Figure 8.1 serves as an overview of the optimization process, the details of which will be presented in the remainder of this chapter.

The input to the optimizer is an entity (formula or term) to be optimized, together with an optimization context providing additional information about the

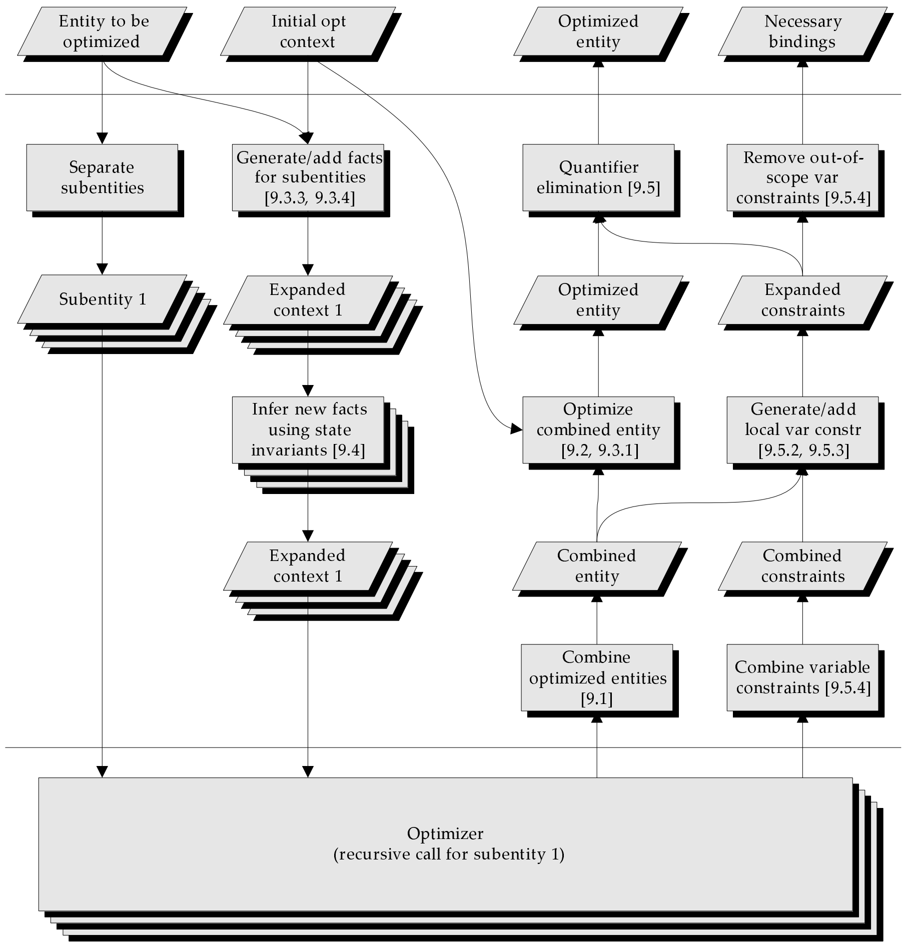

Entity to be optimized
Initial opt context
Optimized entity
Necessary bindings
Separate subentities
Generate/add facts for subentities [9.3.3, 9.3.4]
Quantifier elimination [9.5]
Remove out-of-scope var constraints [9.5.4]
Subentity 1
Expanded context 1
Optimized entity
Expanded constraints
Infer new facts using state invariants [9.4]
Optimize combined entity [9.2, 9.3.1]
Generate/add local var constr [9.5.2, 9.5.3]
Expanded context 1
Combined entity
Combined constraints
Combine optimized entities [9.1]
Combine variable constraints [9.5.4]
Optimizer (recursive call for subentity 1)

Figure 8.1: Control Analysis and Optimization

context in which the entity will eventually be evaluated. The optimizer begins by recursively optimizing all subentities – for example, the conjuncts in a conjunction, or the temporal term and fluent term forming a *value*() expression. Part of this processing is identical for each subentity, indicated as layers of depth in the figure, including the generation of new context information for each subentity.

Each recursive call to the optimizer returns an optimized subentity and, in case the entity is a formula, a set of necessary constraints on variable values. Further optimizations may be applied once the subentities have been combined, and new variable constraints may be generated from the combined entity. If sufficiently strong variable constraints are generated, it may be possible to eliminate quantifiers, reducing the time complexity of formula evaluation (Section 8.5).

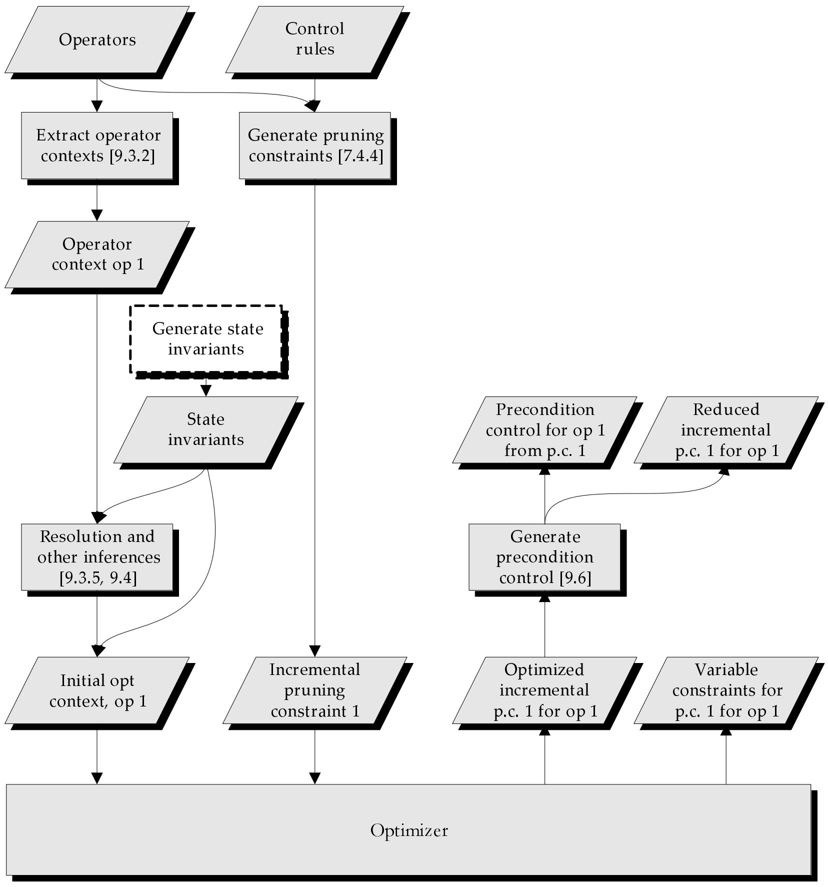

Operators
Control rules
Extract operator contexts [9.3.2]
Generate pruning constraints [7.4.4]
Operator context op 1
Generate state invariants
State invariants
Precondition control for op 1 from p.c. 1
Reduced incremental p.c. 1 for op 1
Resolution and other inferences [9.3.5, 9.4]
Generate precondition control [9.6]
Initial opt context, op 1
Incremental pruning constraint 1
Optimized incremental p.c. 1 for op 1
Variable constraints for p.c. 1 for op 1
Optimizer

Figure 8.2: Control Analysis and Optimization

The optimizer makes use of a number of general techniques applicable to most types of formulas and expressions, but special emphasis has been placed on the optimization of incremental pruning constraints (Figure 8.2). As specified in the previous chapter, each control formula in a narrative generates separate incremental pruning constraints for each operator type in a narrative. Information extracted from an operator definition can then be used to generate an optimization context enhancing the optimization of the associated pruning constraints. Several forms of operator analysis have been developed for this purpose, including the extraction of precondition and effect facts to provide information about the context in which an incremental pruning constraint will be evaluated as well as an analysis of the potential state transitions that may take place during the execution of an operator instance which may aid in the optimization of triggered state transition constraints.

Existing domain analysis techniques have also been considered for inclusion in the formula and term optimization framework. At the moment, the most suitable candidate appears to be the automatic extraction of state invariants. In order to verify the hypothesis that such invariants can indeed be a useful addition to the optimizer, the first step has involved accepting and using manually declared state invariants (Section 8.4).

Applying the optimizer to an incremental pruning constraint can often result in a conjunction where some conjuncts only refer to the invocation timepoint of an operator. Such conjuncts can be moved to the precondition of the operator, which leads to fewer actions being applied and fewer states being expanded (Section 8.6).

These techniques have proven very effective in many domains. As demonstrated by the benchmark tests at the end of the chapter, performance is improved by a factor of 40 for the largest logistics problems from the Second International Planning Competition (IPC-2000, held at the AIPS-2000 conference: Bacchus, 2001) and by a factor of 400 for the largest blocks world problems.

8.1 General Optimization Framework

The general optimization framework developed in this chapter will be applied to $\mathcal{L}(\text{ND})^*$ formulas as well as value terms, temporal terms and fluent terms. Formulas and terms will sometimes simply be referred to as "entities" to be optimized.

An explicit enumeration of all details of all optimizations, as they are applied to each of the numerous types of entities supported by the optimizer, would be quite tedious. Instead, we will generally discuss and explain the underlying principles behind the optimizations. The optimizer function resulting from applying these principles will be denoted by optimize().

Basic recursive optimization. If nothing else is stated, an atomic entity can be optimized to itself, while a composite entity may be optimized by first recursively optimizing its constituent parts and then constructing a new entity of the same type from the optimized parts in the obvious manner. All other optimizations specified below are added on top of this basic framework.

Initial processing. Certain optimizations discussed below may determine that a complex value term in an incremental pruning constraint must always take on the same value as a formal argument variable belonging to the corresponding operator, and will then replace the value term with the variable. Such variables are later substituted with the actual arguments of an action occurrence when the pruning constraint is tested, as specified in Definition 6.4.8 on page 174, removing the need to evaluate the original complex value term. For this procedure to work as intended, the formal argument variable must occur free rather than bound in the optimized formula. To ensure that this will always be the case, all variables bound in an entity are replaced with fresh unused variables of the same sort before optimization begins.

Connectives. Only negations, disjunctions, conjunctions and equivalences are considered by the formula optimizer. Disjunctions and conjunctions are generalized to support arbitrary arities.

Example 8.1.1 (Logistics Domain, continued)
Renaming bound variables in the three incremental pruning constraints for the logistics domain, as shown in Example 6.4.2 on page 177, may result in the following constraints.

For only-load-into-plane-when-necessary:
$\forall obj_{10}, plane_{10}, loc_{10}.$
$[s]\ \neg\mathsf{in}(obj_{10}, plane_{10}) \wedge \mathsf{at}(obj_{10}, loc_{10}) \wedge [s{+}1]\ \mathsf{in}(obj_{10}, plane_{10}) \rightarrow$
$\exists loc'_{10}\ [\ \mathsf{goal}(\mathsf{at}(obj_{10}, loc'_{10})) \wedge [s]\ \mathsf{city\text{-}of}(loc_{10}) \hat{\neq} \mathsf{city\text{-}of}(loc'_{10})\ ]$

For only-unload-from-plane-when-necessary:
$\forall obj_{20}, plane_{20}, loc_{20}.$
$[s]\ \mathsf{in}(obj_{20}, plane_{20}) \wedge \mathsf{at}(plane_{20}, loc_{20}) \wedge [s{+}1]\ \neg\mathsf{in}(obj_{20}, plane_{20}) \rightarrow$
$\exists loc'_{20}\ [\ \mathsf{goal}(\mathsf{at}(obj_{20}, loc'_{20})) \wedge [s]\ \mathsf{city\text{-}of}(loc_{20}) \hat{=} \mathsf{city\text{-}of}(loc'_{20})\ ]$

For objects-remain-at-destinations:
$\forall obj_{30}, loc_{30}.$
$[s]\ \mathsf{at}(obj_{30}, loc_{30}) \wedge \mathsf{goal}(\mathsf{at}(obj_{30}, loc_{30})) \rightarrow [s{+}1]\ \mathsf{at}(obj_{30}, loc_{30})$ ■

8.2 Equivalence Optimizations

The first type of optimization performed by TALplanner involves rewriting a formula α to a simpler form β such that $\alpha \equiv \beta$. TALplanner implements a number of such optimizations, making use of well-known logical equivalences such as $\phi \wedge (\phi \vee \psi) \equiv \phi$, $\phi \wedge \mathtt{true} \equiv \phi$, and $\forall x.(\alpha \wedge \beta(x)) \equiv \alpha \wedge \forall x.\beta(x)$ where x does not occur in α.

At first glance it would appear that the only use for this type of optimization would be to correct trivial mistakes made by the domain designer. After all, who would deliberately write a formula such as $[t]\ \mathsf{on}(x,y) \wedge ([t]\ \mathsf{on}(x,y) \vee [t]\ \mathsf{ontable}(x))$? The answer lies in the fact that TALplanner conjoins incremental pruning constraints originating from different control rules, the optimizer performs various types of simplifications, and the optimizer may generate new preconditions to be conjoined to existing preconditions. At various points in this process, the formulas that are generated may be susceptible to equivalence optimizations that were not applicable to the original formulas. Performing these optimizations can therefore have a noticeable impact on the performance of the planner.

Additionally, formulas in a planning domain may quantify over variables that in certain planning problems turn out to have empty value domains, in which case $\forall v.\phi(v)$ is optimized to $\mathtt{true}$ and $\exists v.\phi(v)$ is optimized to $\mathtt{false}$. For singleton value domains, the formula $\forall v.\phi(v)$, where the domain of v contains the single value $\mathbf{w}$, is optimized to $\phi(\mathbf{w})$, and similarly for existential quantification. Also, any value term whose associated value domain contains a single value $\mathbf{w}$ must necessarily take on that value.

8.3 Context-Dependent Optimizations

The potential for optimizations can be vastly extended by considering the context in which an entity will be evaluated.

Consider again a formula α that should be optimized, and suppose that it can somehow be determined in advance that α will only be evaluated in a context where another formula γ is satisfied. For example, maybe α is intended to be used as an incremental pruning constraint for a specific operator, in which case it will be evaluated immediately after applying an instance of that operator, which may guarantee that certain conditions hold. In this case, the optimizer is not limited to finding a simpler but equivalent formula β such that $\alpha \equiv \beta$. Instead, it can search for a simpler and potentially *weaker* formula β such that $(\alpha \wedge \gamma) \equiv (\beta \wedge \gamma)$. For any narrative where γ holds, α will always be true iff β is true, even though this is not necessarily the case in narratives where γ does not hold. Given that γ is sufficiently strong, this clearly provides far better optimization opportunities than pure equivalence optimization.

As a second step towards improving the efficiency of formula evaluation, the TALplanner formula optimizer therefore makes use of the context in which a formula will eventually be evaluated. The formula optimizer is extended to take two arguments: A formula α to be optimized and an optimization context.

Definition 8.3.1 (Optimization Context)
An *optimization context*[1] is a tuple containing the following elements:

- A set of formulas Φ known to hold when an entity will be evaluated.
- If an incremental pruning constraint in incr_i should be optimized, the operator type o^i associated with the constraint. ■

If an operator type is specified, its formal arguments may occur free in α as well as in the formula returned by the optimizer. Before evaluation, these variables will be instantiated with the actual arguments of the particular operator instance of that had just been applied, as specified in Definition 6.4.8 on page 174.

Below, we will explain how context information is generated automatically by the optimizer and how this context information can be used to simplify formulas and terms. The presentation assumes the optimization context $\langle \Phi, o \rangle$ does contain an operator type. If it does not, those parts of the optimizer that depend on the operator type are deactivated.

[1]Redefined in Definition 8.4.1 on page 221 to add support for state invariants.

8.3.1 Using Context Information

Context information is used in the optimization of atomic formulas, where an entailment checker attempts to determine whether an atomic formula is entailed by the context Φ (in which case it can be optimized to `true`) or whether its negation is entailed (`false`). It should be noted that although this entailment checker must be sound it need not be complete. Incompleteness weakens the optimizer but does not affect correctness.

Context information is also used when optimizing terms. For example, suppose that the optimizer is currently optimizing the complex value term ω, and suppose that a formula in the optimization context has the form $v = \omega$ or $\omega = v$ for some value variable v. Value variables are evaluated more quickly than complex value terms, which implies that ω should be replaced with v – as long as this does not violate sort constraints. For example, depending on the context in which a `car`-valued term occurs, it cannot necessarily be replaced with a `vehicle`-valued variable, where `vehicle` is a strict supersort of `car`.

Similarly, context information can be used in the optimization of *Holds* formulas: Given the knowledge that $[\tau]\ f \mathrel{\hat{=}} v$, the formula $[\tau]\ f \mathrel{\hat{=}} \omega$ holds iff $v = \omega$.

8.3.2 Generating Initial Context using Operator Analysis

If a formula to be optimized is an incremental pruning constraint, it is possible to provide a certain amount of context even for the initial call to the optimizer.

Incremental pruning constraints should mainly depend on the state or states generated by the latest operator invocation, and although the preprocessor cannot know in advance which operator instance was invoked, it *can* know which operator *type* was invoked (such as drive or fly in the logistics domain) – this is part of the reason why there is a separate set of constraints incr_i for each operator type o^i (Section 6.4.3). This leads to the idea of extracting some context information from the operator definitions regarding the states in which the constraints will be evaluated.

If the precondition of an operator is not satisfied, the operator instance is never applied. If it is satisfied, the effects are applied, and if they are inconsistent, the planner backtracks before testing incremental pruning constraints. In other words, the incremental pruning constraints in incr_i are only tested if both the precondition and the effects of the corresponding operator o^i are known to hold. Consequently, a set of known facts can be extracted from the operator quite easily. Let ϕ be the precondition of o^i and let ϕ' be a conjunction of fixed fluent formulas extracted from the unconditional effects of the action (for example, the effect $[s+1]\ \mathsf{at}(o,l) :=$ **false** generates the formula $[s+1]\ \mathsf{at}(o,l) \mathrel{\hat{=}}$ **false**). Then, the initial set of known formulas is $\phi \wedge \phi'$.

Example 8.3.1 (Operator Analysis for the Logistics Domain)
Analyzing the drive operator from the logistics domain (Example 6.2.10 on page 157)

yields the formulas $\phi = [s]\ \mathsf{at}(\mathit{truck}, \mathit{loc}_1) \wedge \mathsf{city\text{-}of}(\mathit{loc}_1) \mathrel{\hat{=}} \mathsf{city\text{-}of}(\mathit{loc}_2) \wedge \mathit{loc}_1 \neq \mathit{loc}_2$ and $\phi' = [s+1]\ \mathsf{at}(\mathit{truck}, \mathit{loc}_1) \mathrel{\hat{=}} \mathbf{false} \wedge [s+1]\ \mathsf{at}(\mathit{truck}, \mathit{loc}_2) \mathrel{\hat{=}} \mathbf{true}$. ■

Note that both the formal invocation timepoint of the operator o^i and its formal arguments can occur as free variables in ϕ or ϕ'. When the constraints in incr_i are tested, the formal arguments will be instantiated with the values used during the latest operator invocation, as stated in the definition of incr_i (Section 6.4.3). In this way, an incremental pruning constraint can refer directly to the arguments of the corresponding operator invocation. (Since all variables in control rules have been replaced with fresh variables, there is no risk of mistaking one instance of a variable for another.)

8.3.3 Passing On Context

The optimization context $\langle \Phi, o \rangle$ given to $\mathsf{optimize}()$ is generally passed on unmodified when the optimizer makes a recursive call to optimize a subformula or subterm. Some exceptions where new context information can be generated will be discussed below.

8.3.4 Generating Context from Composite Formulas

For certain types of composite formulas, the optimizer is not limited to passing on the initial context but can generate new context information for use in the optimization of subformulas.

For a conjunction $\bigwedge_{i=0}^{n} \phi_i$, the value of any single conjunct ϕ_k is irrelevant if any other conjunct is false, because in this case the conjunction will definitely be false. Conversely, the value of the conjunct ϕ_k is only relevant in the case where all other conjuncts are true. Therefore, the optimizer recursively optimizes each ϕ_k in a context where all other conjuncts hold, that is, where $\bigwedge_{0 \leq i \leq n, i \neq k} \phi_i$ has been added to Φ. Conjoining the resulting optimized subformulas results in a new formula which is guaranteed to be equivalent to the original conjunction in the given context, even if each optimized subformula taken in separation may not necessarily be equivalent to the corresponding original conjunct. An example will be given in Section 8.3.6 below.

A dual optimization is applied to disjunctions.

8.3.5 Inferring Additional Context

The context information generated above can be augmented by an automatic inference procedure $\mathsf{infer}(\Phi)$ which generates new facts from an existing set of facts Φ. The return value is a set of formulas containing the original facts Φ and possibly additional formulas that are entailed by $\mathit{Trans}^{+}(\Phi)$.

The first version of the inference procedure uses standard equivalences to generate new facts from Φ. This reduces the amount of work done by the entailment checker when testing whether an atomic formula is entailed by a given context.

Definition 8.3.2 (Inference Procedure)
The TALplanner *inference procedure*[2] $\mathsf{infer}(\Phi)$ is defined by repeatedly applying the following inference rules to the formulas in Φ and adding any newly inferred formulas to Φ until a fixpoint is reached.

- Given $[\tau]\ f \hat{=} \omega$, infer $value(\tau, f) = \omega$, and vice versa.
- Given $\omega_1 = \omega_2$ and $\omega_2 = \omega_3$, infer $\omega_1 = \omega_3$.
- Given $\neg(\alpha \vee \beta)$, infer $\neg\alpha$ and $\neg\beta$.
- Given $\forall v.\alpha \wedge \beta$, infer $\forall v.\alpha$ and $\forall v.\beta$.
- Given $\neg\forall v.\alpha$, infer $\exists v.\neg\alpha$.
- Given $\neg\exists v.\alpha$, infer $\forall v.\neg\alpha$. ■

8.3.6 Optimization Example

The following example illustrates some of the optimizations described above.

Example 8.3.2 (Optimizing Pruning Constraints)
Suppose that while processing pruning constraints for the logistics domain, the formula $city = value(t, \mathsf{city\text{-}of}(loc_1)) \wedge [t]\ \mathsf{city\text{-}of}(loc_1) \hat{=} value(t, \mathsf{city\text{-}of}(loc_2))$ is generated, possibly due to conjoining multiple control formulas with different origins. As specified in Section 8.3.4, each conjunct in this conjunction will be optimized under the assumption that the other conjunct holds. This entails optimizing the formula $[t]\ \mathsf{city\text{-}of}(loc_1) \hat{=} value(t, \mathsf{city\text{-}of}(loc_2))$ under the assumption that $city = value(t, \mathsf{city\text{-}of}(loc_1))$ holds. Applying the TALplanner inference procedure from Section 8.3.5 to this assumption allows the optimizer to infer that $[t]\ \mathsf{city\text{-}of}(loc_1) = city$ must hold. As stated in Section 8.3.1, the optimizer can now optimize the first conjunct $[t]\ \mathsf{city\text{-}of}(loc_1) \hat{=} value(t, \mathsf{city\text{-}of}(loc_2))$ to the simpler formula $city = value(t, \mathsf{city\text{-}of}(loc_2))$. Though this new conjunct is not equivalent to the original first conjunct, the resulting complete formula $city = value(t, \mathsf{city\text{-}of}(loc_1)) \wedge city = value(t, \mathsf{city\text{-}of}(loc_2))$ is equivalent to the original complete formula. ■

8.4 Using State Invariants

State invariants are conditions that hold in all states generated by any executable plan. In the blocks world, a block that is being held is never clear, and a block that

[2]Redefined in Definition 8.4.2 on page 222 to add support for state invariants.

is ontable is never on another block, though this is never stated explicitly in the domain model but is implicit in the operator definitions and in implicit constraints on the initial state. In the standard logistics domain, a package which is in a vehicle is never at a location: $\forall t, obj_1, vehicle_1, loc_1.[t]\ \mathsf{in}(obj_1, vehicle_1) \rightarrow \neg\mathsf{at}(obj_1, loc_1)$. (While this may appear counter-intuitive, it does follow from the way the logistics domain is usually modeled.) Similar state invariants can be found for most planning domains, but up to now there was no natural place for these constraints to be used in TALplanner.

The optimization techniques presented in the preceding section are based on the use of information about the context in which a formula or term is evaluated. It is clear that the more context information the optimizer has as its disposal, the better the optimization opportunities – and state invariants can be used to infer new information from existing facts. Better yet, there are already automated domain analysis techniques in the literature that extract such constraints from domain definitions (Fox & Long, 1998; Gerevini & Schubert, 1998, 2000; Scholz, 2000; Rintanen, 2000b). Thus we return to the original idea of using existing domain analysis techniques to improve the performance of TALplanner, though with a novel use of the information extracted by these techniques.

There are two steps involved in integrating an automated state invariant extraction algorithm into the planner: The algorithm must be adapted to work with TALplanner's operator definitions (and possibly extended to handle operators with extended duration), and the planner must be altered to actually use the state invariants once they have been generated. We have chosen to begin with the second step, extending TALplanner to make use of *manually* specified state invariants. This will provide the opportunity to test carefully whether the use of the invariants has a sufficient impact on the planner's performance to warrant following through with the implementation of the automatic domain analysis. As in all formulas specified as input to the optimizer, all variables bound in state invariants are replaced with fresh unused variables of the same sort before optimization begins.

The optimization context used by the formula optimizer, previously specified in Definition 8.3.1 on page 217, is extended as follows.

Definition 8.4.1 (Optimization Context)
An *optimization context* is a tuple containing the following elements:

- A set of formulas Φ known to hold when an entity will be evaluated.
- A set of state invariants Ψ that must always hold in all problem instances for the current planning domain
- For incremental pruning constraints, the operator type o associated with the constraint. ■

The inference procedure from Definition 8.3.2 on the facing page is extended to take two arguments: $\mathsf{infer}(\Phi, \Psi)$ generates new facts from an existing set of facts Φ and a set of state invariants Ψ known to hold in all problem instances for the current planning domain. The return value is a set of formulas containing the original facts Φ and possibly additional formulas that are entailed by $\mathit{Trans}^{+}(\Phi \wedge \Psi)$.

In addition to using the standard equivalences shown in the previous definition, facts are also combined with state invariants with limited use of a resolution algorithm. This may yield further facts to be added to Φ, strengthening the information available to the optimizer. For example, the unload operator in the logistics domain provides the context fact $[s]\ \mathsf{in}(obj, vehicle)$. Combining this with the invariant $\forall t, obj_1, vehicle_1, loc_1.[t]\ \mathsf{in}(obj_1, vehicle_1) \rightarrow \neg\mathsf{at}(obj_1, loc_1)$ would generate the formula $\forall loc_1.[s]\ \neg\mathsf{at}(obj, loc_1)$.

Definition 8.4.2 (Inference Procedure)
The extended TALplanner inference procedure $\mathsf{infer}(\Phi, \Psi)$ is defined by repeatedly (1) applying the inference rules from Definition 8.4.2 to the formulas in Φ and (2) applying a resolution inference procedure to combine facts in Φ with invariants in Ψ, adding any newly inferred formulas to Φ, until a fixpoint is reached. ■

The performance impact of state invariants can be expected to vary depending on the degree of correlation between state variables. In the standard four-operator formulation of the blocks world, for example, state variables are highly correlated: A block which is on another block is never on the table, a block which has another block on top of it is never clear, the hand is empty if and only if no block is being held, and a block which is being held is never clear, on top of another block, or on the table. Because of this high degree of redundancy, the same facts can be expressed in many different syntactic forms. State invariants make the correlations between state variables explicit, which makes the remaining domain analysis algorithms considerably more robust against these syntactic variations. As can be seen in the benchmark tests later in this chapter, the use of state invariants decreases the time required to solve some blocks world problem instances by a factor of 3. In other domains, state variables may be less correlated, which naturally decreases the impact of using state invariants.

A future version of TALplanner may be integrated with an automatic analysis method to avoid the need to specify state invariants as part of the planning domain description.

8.5 Eliminating Quantifiers

Since TAL uses finite value domains, any universally quantified formula $\forall x.\phi(x)$ can be evaluated simply by iterating over each possible value of the variable x. However, iterating over a large set of values is quite inefficient. If it can be determined in advance that $\phi(x)$ can only be false if x belongs to some smaller set of value terms X (and must be true for all other values of x), then it is sufficient to verify the formula $\forall x \in X.\phi(x)$. A dual optimization can be applied to existentially quantified formulas $\exists y.\phi(y)$, if it can be determined that $\phi(y)$ can only be true if y belongs to some set of value terms Y. If X (or Y) is a singleton, this leads to the complete elimination of a quantifier.

The formula optimizer used by TALplanner is extended to return not only an optimized formula ψ but a tuple $\langle \psi, \mathsf{necNeg}, \mathsf{necPos}\rangle$, where necNeg is a set of variable constraints necessary for $\neg\psi$ to hold (corresponding to X above), and necPos is a set of variable constraints necessary for ψ to hold (corresponding to Y above). Each variable constraint in necNeg (necPos) has the form $x \mapsto \{\omega_1, \ldots, \omega_n\}$ indicating that ψ can only be false (true) if $\bigvee_{i=1}^{n} x = \omega_i$.

Variable constraints (Section 8.5.1) are generated by certain atomic expressions (Section 8.5.2), and are propagated and combined upwards through the call chain as required by connectives and quantifiers. Variable constraints can also be generated using a form of state transition analysis (Section 8.5.3).

8.5.1 Variable Constraints

Before discussing how variable constraints are generated and used, we must provide formal definitions of constraints and constraint sets, and a number of operations on these structures.

Definition 8.5.1 (Variable Constraint)
A *variable constraint* for the value variable x is an expression $x \mapsto \{\omega_1, \ldots, \omega_n\}$ where each ω_i is a value term, denoting that x must take on the same value as one of the value terms in $\{\omega_1, \ldots, \omega_n\}$. An *inconsistent variable constraint* for the value variable x is represented as the expression $x \mapsto \emptyset$. ■

The disjunction of two variable constraints consists of the union of all possible values according to either constraint.

Definition 8.5.2 (Variable Constraint Disjunction)
Let $x \mapsto \{\omega_1, \ldots, \omega_m\}$ and $x \mapsto \{\omega'_1, \ldots, \omega'_n\}$ be two variable constraints for the same value variable x. Then, the disjunction of these variable constraints, denoted by $(x \mapsto \{\omega_1, \ldots, \omega_m\}) \vee (x \mapsto \{\omega'_1, \ldots, \omega'_n\})$, is $x \mapsto \{\omega_1, \ldots, \omega_m, \omega'_1, \ldots, \omega'_n\}$. ■

Conjunction of variable constraints is slightly more complicated. Each variable constraint maps a variable to a set of value terms. This set does not necessarily consist of value constants – for example, it can contain value variables whose eventual bindings are not known during optimization. A straight intersection between two constraints $x \mapsto \{y\}$ and $x \mapsto \{z\}$ would then result in the inconsistent constraint $x \mapsto \emptyset$, which is unsound because even though y and z are distinct variables they may eventually be bound to the same value.

Instead of retaining full information for conjoined variable constraints, we use a weaker representation where some information is lost but where the representation remains small in size.

Definition 8.5.3 (Variable Constraint Conjunction)
Let $x \mapsto \{\omega_1, \ldots, \omega_m\}$ and $x \mapsto \{\omega'_1, \ldots, \omega'_n\}$ be two variable constraints for the same value variable x. Then, the conjunction of these variable constraints, denoted by $(x \mapsto \{\omega_1, \ldots, \omega_m\}) \wedge (x \mapsto \{\omega'_1, \ldots, \omega'_n\})$, is constructed as follows.

Let V be the set of value name constants[3] in $\{\omega_1, \ldots, \omega_m\}$, and let W be the remaining value terms in that set. Let V' be the set of value name constants in $\{\omega'_1, \ldots, \omega'_n\}$, and let W' be the remaining value terms in that set. The conjunction of the two variable constraints is $x \mapsto (V \cap V') \cup W \cup W'$. ■

An example may be in order.

Example 8.5.1 (Variable Constraint Disjunction and Conjunction)
Let $loc \mapsto \{\mathbf{loc1}, \mathbf{loc3}, loc_2, value(t, \mathsf{location}(\mathbf{truck7}))\}$ and $loc \mapsto \{\mathbf{loc1}, \mathbf{loc2}, loc_2\}$ be two variable constraints. The disjunction of these variable constraints is $loc \mapsto \{\mathbf{loc1}, \mathbf{loc2}, \mathbf{loc3}, loc_2, value(t, \mathsf{location}(\mathbf{truck7}))\}$: Each of the value terms represented by the original variable constraints is a possible binding for *loc*.

Now consider the conjunction of the same two constraints. This should generate an overestimate of the set of values that *loc* could take on given that both variable constraints are satisfied. The value terms **loc1** and loc_2 are present in both constraints, and should definitely be part of the overestimate. The value term **loc3** is only explicitly present in the first constraint, but could potentially denote the same value as loc_2 in the second constraint, so at least one of these terms has to be present. Similarly, **loc2** is only explicitly present in the second constraint, but could denote the same value as loc_2 or $value(t, \mathsf{location}(\mathbf{truck7}))$ in the first constraint, so either **loc2** or both loc_2 and $value(t, \mathsf{location}(\mathbf{truck7}))$ must be present. Finally, $value(t, \mathsf{location}(\mathbf{truck7}))$ is only explicitly present in the first constraint, but could take on the same value as *any* value term in the second constraint, so either this term or *all* value terms in the second constraint must be present.

The definition above generates the sets of value terms $V = \{\mathbf{loc1}, \mathbf{loc3}\}$, $W = \{loc_2, value(t, \mathsf{location}(\mathbf{truck7}))\}$, $V' = \{\mathbf{loc1}, \mathbf{loc2}\}$, and $W' = \{loc_2\}$. This yields the final result $loc \mapsto \{\mathbf{loc1}, loc_2, value(t, \mathsf{location}(\mathbf{truck7}))\}$, which satisfies all conditions given in the previous paragraph. ■

Variable constraints can be combined into sets.

Definition 8.5.4 (Variable Constraint Set)
A *variable constraint set* is a possibly empty set of variable constraints for distinct value variables. A variable constraint set *binds* a value variable x iff the variable constraint set contains a variable constraint for x.

Let c be a variable constraint set and x a value variable. If c binds x, then $c[x]$ denotes the unique variable constraint for x in c. Otherwise, c does not bind x, and

[3]Value name constants are those value constants that are declared to belong to a value domain. Due to the use of unique name axioms, two distinct value name constants are guaranteed to denote distinct values.

$c[x]$ denotes the (unconstraining) variable constraint $x \mapsto X$ where X contains all values from the domain of x.

An *inconsistent variable constraint set* is a variable constraint set containing at least one inconsistent variable constraint.

The *conjunction* of two variable constraint sets c and c' is
$\{c[x] \wedge c'[x] \mid x \text{ is bound in } c \text{ or } c'\}$.

The *disjunction* of two variable constraint sets c and c' is
$\{c[x] \vee c'[x] \mid x \text{ is bound in } c \text{ or } c'\}$. ■

8.5.2 Generating Constraints from Atomic Formulas

Variable constraints can be generated by equality expressions: Optimizing the formula $x = \omega$ generates the constraint $x \mapsto \{\omega\}$ in necPos while leaving necNeg empty, and $x \neq \omega$ generates the variable constraint $x \mapsto \{\omega\}$ in necNeg while leaving necPos empty.

Similarly, a fixed fluent formula $[\tau]\ f \mathrel{\hat{=}} x$ generates a positive variable constraint $x \mapsto \{value(\tau, f)\}$, and a fixed fluent formula $[\tau]\ f \mathrel{\hat{\neq}} x$ generates a negative variable constraint $x \mapsto \{value(\tau, f)\}$.

Optimizing $[\tau]\ f \mathrel{\hat{=}} \omega$ with a known formula $[\tau]\ f \mathrel{\hat{=}} x$ in the optimization context generates a positive variable constraint $x \mapsto \{\omega\}$. If exactly one of these formulas is negated, a negative variable constraint $x \mapsto \{\omega\}$ is generated instead, while if both formulas are negated, a positive constraint is generated.

8.5.3 Generating Constraints using State Transition Analysis

Many control rules can only be violated if certain state transitions take place. This is a natural consequence of the fact that many control rules follow a certain pattern, where a property true in one state should either be preserved to the next state or violated in a very specific way.

For example, only-load-into-plane-when-necessary (initially shown in Example 6.4.1 on page 168) generates incremental pruning constraints (shown in Example 8.1.1) that state that a package should only be loaded into a plane if a plane is needed to move it. This can also be stated in another way: If a package is not in a plane in a certain state, then it should *remain* not in that plane in the next state, *unless* a plane is needed in order to move it. As long as the property $\neg\mathsf{in}(obj_{10}, plane_{10})$ is preserved from s to $s+1$ for all obj_{10} and all $plane_{10}$, the constraint cannot be violated.

As another example, the incremental pruning constraints generated by objects-remain-at-destinations state that if a package is at a certain location at time s, it must *remain* at that location at $s+1$, *unless* there is no goal that it should remain there. If the property $\mathsf{at}(obj_{30}, loc_{30})$ is preserved from s to $s+1$ for all obj_{30} and all loc_{30}, the constraint cannot be violated.

Clearly, it would be a major advantage if the preprocessor could determine in advance that these state transitions cannot take place – then, the entire incremental constraints would necessarily be true, and would not need to be tested. Failing this, it would be of almost equal benefit to the planner if it could be determined that the state transitions can only take place for certain specific instances of a fluent, thereby generating additional variable constraints and reducing the number of instances of an incremental constraint that need to be tested. In fact, this *can* be detected in advance, as will be demonstrated in the following example.

Example 8.5.2 (State Transition Analysis)
Returning to objects-remain-at-destinations, the incremental pruning constraints generated by this rule can only be violated if an instance of $\mathsf{at}(obj_{30}, loc_{30})$ is made false between s and $s+1$. But s is the invocation timepoint of the latest operator, and $s+1$ is time of the effect state.

The unload operator never makes an instance of at false at $s+1$, and therefore this incremental pruning constraint is never violated for unload.

Although drive makes $\mathsf{at}(vehicle, loc_1)$ false, this instance refers to the location of a vehicle rather than that of an object and cannot be unified with $\mathsf{at}(obj_{10}, loc_{10})$. Therefore, the incremental constraint can never be violated by drive.

The load action makes $\mathsf{at}(obj, loc)$ false, and unifying this with $\mathsf{at}(obj_{10}, loc_{10})$ yields the variable constraints $obj_{10} \mapsto \{obj\}$ and $loc_{10} \mapsto \{loc\}$. These variable constraints must necessarily hold if the disjunction should be false, and can therefore be added to necNeg when the disjunction is analyzed. ■

These insights can be used to improve the formula optimizer.

Extending the Optimizer

The following algorithm is called from the optimizer when analyzing a disjunction, given the disjunction and an optimization context as arguments. The return value is a variable constraint set necNeg that is required for the disjunction to be false: If any of the variable constraints in necNeg does not hold, the disjunction will be satisfied. Explanations will be provided below.

```
1  procedure find-necessary-constraints($\bigvee_{i=1}^{n} \phi_i$, $\langle \Phi, \Psi, o \rangle$)
2  let conjuncts = infer($\Phi \wedge \bigwedge_{i=1}^{n} \neg\phi_i, \Psi$)        Add negation of disjunction to $\Phi$
3  let necNeg = $\emptyset$
4  for all $[\tau]\ f \hat{=} \omega$ in conjuncts do
5    for all $[\tau']\ f \hat{=} \omega'$ in conjuncts do                       Identical f!
6      if can prove $\tau < \tau'$ then
7        if can prove that $\omega$ and $\omega'$ cannot take on the same value then
8          if can prove that $\tau \geq t_{\mathsf{max}}$ then
9            return an impossible binding
10         if sequential operator type $o$ given then
11           let necNeg = necNeg$\wedge$ analyzeST($[\tau]\ f \hat{=} \omega$, $[\tau']\ f \hat{=} \omega'$, $\langle \Phi, \Psi, o \rangle$)
12 return necNeg
```

The incremental pruning constraint for objects-remain-at-destinations relative to load-plane is

$\forall obj_{30}, loc_{30}.\ [s]\ \mathsf{at}(obj_{30}, loc_{30}) \wedge \mathsf{goal}(\mathsf{at}(obj_{30}, loc_{30})) \rightarrow [s{+}1]\ \mathsf{at}(obj_{30}, loc_{30})$

where the implication inside the universal quantifier prefix can also be written as a disjunction $\bigvee_{i=1}^{n} \alpha_i$. This disjunction can be analyzed using the algorithm above.

For the disjunction to be false, it must clearly be the case that $\bigwedge_{i=1}^{n} \neg\alpha_i$. There is also a set of formulas Φ that are known to hold regardless of whether the disjunction holds or not, so for the disjunction to be false, we must have $\Phi \wedge \bigwedge_{i=1}^{n} \neg\alpha_i$. The resolution inference algorithm can be used together with the state invariants to infer additional facts: For the disjunction to be false, $\mathsf{infer}(\Phi \wedge \bigwedge_{i=1}^{n} \neg\alpha_i, \Psi)$ must hold. For example, since it must be the case that $[s+1]\ \mathsf{at}(obj_{30}, loc_{30})$, it is possible to infer $\forall vehicle.[s+1]\ \neg\mathsf{in}(obj_{30}, vehicle)$. The conjunction of the formulas returned by infer will be denoted by $\bigwedge_{i=1}^{m} \beta_i$.

Now, suppose that β_i is $[\tau]\ f \hat{=} \omega$ and that β_j is the formula $[\tau']\ f \hat{=} \omega'$. Suppose further that it can be proven[4] that $\tau < \tau'$, so the second formula refers to a later timepoint, and that $\omega \neq \omega'$. Due to β_i, the fluent could not have taken on the value ω' at τ, but due to β_j, it must take on that value at τ'. The value of f must have changed in the interval $(\tau, \tau']$.[5]

What remains is trying to find a set of variable constraints that are necessary for f to be able to change in $(\tau, \tau']$, or in the best case, to determine that f in fact must remain constant. If any such constraints are found, they can be conjoined to necNeg, since the constraints are necessary for the disjunction to be false. TALplanner uses two different types of state transition analysis for finding variable constraints.

First, if $\tau \geq t_{\mathsf{max}}$, then the entire interval $(\tau, \tau']$ is strictly after t_{max}. But no effects can take place after t_{max}, so no fluents can change there. Therefore, it is impossible that the disjunction does not hold, and an inconsistent variable constraint set is returned. This is useful for analyzing final constraints, the only constraints that can contain the internal t_{max} constant.

Second, if an operator type o is specified, the disjunction will be evaluated immediately after an operator of that type is invoked, and the transitions possible during the execution interval can be analyzed. This analysis is useful for incremental pruning constraints in incr_i, and is described in detail below.

State Transition Analysis for Sequential Operators

The state transition analysis algorithm for sequential operators is as follows.

[4]Whenever we say "if we can prove ϕ" rather than "if ϕ is the case", failing to prove this fact may lead to a decrease in performance but is always safe. For example, the attempt to prove that $\tau < \tau'$ could be a test whether τ' is of the syntactic form $\tau + n$ for some positive n, or could be a stronger test involving more complex temporal reasoning.

[5]TALplanner also handles negated formulas $\neg[\tau]\ f \hat{=} \omega$ and $\neg[\tau]\ f' \hat{=} \omega'$. The extension is trivial and is omitted to improve the clarity of the presentation.

```
procedure analyzeST($[\tau]\ f \hat{=} \omega$, $[\tau']\ f \hat{=} \omega'$, $\langle \Phi, \Psi, o \rangle$)
if can prove $\tau' > \mathsf{inv}(o)$ then
  let eff = all conditional and unconditional effects of $o$
  for all $[\overline{\tau}]\ g := \omega''$ in eff do
    if can prove $f \neq g$ then remove this from eff
    elsif can prove $\overline{\tau} \notin (\tau, \tau']$ then remove this from eff
    elsif can prove $\omega'$ and $\omega''$ cannot be equal then remove this from eff
  if {free variables in eff} $\subseteq$ {arguments of $o$} then
    let necNeg $= \{x \mapsto \emptyset \mid x$ is free in eff$\}$          Cannot violate the formula...
    for all $[\overline{\tau}]\ g := w$ in eff do
      let necNeg = necNeg$\vee$ unify($g$, $f$)    ...unless unified with some remaining effect
    return necNeg
return $\emptyset$
```

This algorithm returns a set of variable constraints that are required for f to change values from ω to ω' between τ and τ', given that an instance of o is the last operator to be invoked in the current search node. Note that f might be a fluent expression with arguments, such as $\mathsf{at}(obj_1, loc_1)$.

Given the assumption that the last action in the current plan is an instance of o, the only changes that can take place in $(\mathsf{inv}(o), \infty)$ are those explicitly caused by this instance of o. No information is provided about what might have happened in $[0, \mathsf{inv}(o)]$, though, so if it cannot be proven that $\tau' > \mathsf{inv}(o)$, the analysis is aborted.

Otherwise, consider every effect of the operator, conditional as well as unconditional. For load, this would be the two effects $[s+1]$ $\mathsf{at}(obj, loc) :=$ **false** and $[s+1]$ $\mathsf{in}(obj, vehicle) :=$ **true**, where the analyzer has no information about what the formal operator arguments *obj*, *loc* and *vehicle* will be bound to when the formula will be evaluated.

If an effect cannot affect f, it is irrelevant and can be discarded. If it might affect f but not at an interesting timepoint (in the interval $(\tau, \tau']$, when the change must take place), it can be discarded. Finally, if the new value assigned to f by this effect is a value term ω'' that cannot take on the same value as ω', then the effect definitely cannot cause a transition from ω to ω', and can be discarded.

The remaining effects of o might cause f to change values from ω to ω' between τ and τ'. Because the change we are interested in happened strictly after the invocation of o, and because o is assumed to be the last operator in a sequential plan, these effects must also be the *only* possible causes for f to change values in this specific way.

If the remaining effects contain free variables that are not formal argument variables for o, then those variables must have been bound in quantified effects, and the analysis is aborted. Otherwise, it is safe to claim that f must be equal to one of the fluents that were affected by the action. This means it must be unified with one of them for the desired state transition to occur, so the disjunction of all $\mathsf{unify}(g_i, f)$ can be returned.

State Transition Analysis for Concurrent Operators

In order to generate a set of variable bindings necessary for a certain state transition to take place, the state transition analysis algorithm above must be able to determine the complete set of possible effects within the temporal interval when the state transition should occur. This is the only place where the algorithm is dependent upon the fact that the specified operator o is the last operator in a sequential plan.

It would be quite useful if a similar algorithm could be found for concurrent operators. At first glance, this would appear difficult, because when analyzing an incremental pruning constraint relative to one specific operator there is no way of determining what other operators may be invoked concurrently by the planner, which means that it is not possible to determine in advance a complete set of possible state transitions. The key to the solution is proper blame assignment: In concurrent TALplanner, each incremental pruning constraint is tested not once per timepoint but once for each action in the plan. If an incremental pruning constraint can only be violated by a given state transition, it only needs to be tested by the action responsible for causing that state transition. Consequently, the planner can analyze each constraint relative to each operator type under the assumption that this operator is in fact the only cause of state transitions, knowing that control rule violations caused by state transitions that result from other concurrent actions will be handled by the incremental pruning constraints associated with those actions.

8.5.4 Combining Constraints from Composite Formulas

When optimizing a formula of the form $\neg\phi$, the inner formula ϕ is recursively optimized and the generated constraint sets necPos and necNeg are swapped.

When optimizing a conjunction $\bigwedge_{i=0}^{n} \phi_i$, each conjunct is recursively optimized. Denote the return values by $\langle \psi_i, \mathsf{necNeg}_i, \mathsf{necPos}_i \rangle$ for $0 \leq i \leq n$. For the conjunction to hold, the conjunction of all necPos_i must hold; for the conjunction to be false, the disjunction of all necNeg_i must hold. When conjoining all necPos_i, it may be the case that there is no potential common binding for a certain variable (for example, because two conjuncts require bindings that cannot belong to the same value domain). In this case, the resulting variable constraint set is inconsistent, and the formula may be immediately optimized to `true` or `false`.

A dual optimization is applied to disjunctions.

For a quantified formula $\forall x.\phi$ or $\exists x.\phi$, the variable constraint sets generated for the inner formula are returned after removing any constraints for the quantified variable x, which is not in scope outside the quantified formula. This does not require iteration over possible values of x.

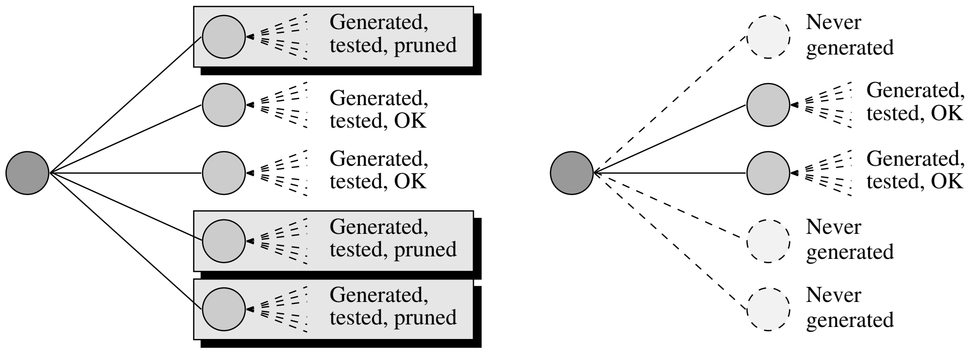


Figure 8.3: Fewer States Generated using Precondition Control

8.6 Generating Precondition Control

In order to test whether an instance of the operator o^i violates one of its associated incremental control formulas in incr_i, TALplanner generally has to begin by adding this operator instance to the current plan candidate and calculating the new states generated by the new operator instance. Only then can the formulas in incr_i be evaluated in the new states (the left hand side of Figure 8.3).

However, after applying domain analysis and the associated optimizations to incr_i, the resulting formulas often turn out to have conjuncts that only refer to the invocation timepoint of o^i. Because these conjuncts do not depend on the states generated by applying an operator instance, they can be moved from incr_i into the precondition of the operator.

In the logistics domain, for example, the precondition $\exists loc'[\mathsf{goal}(\mathsf{at}(obj, loc')) \wedge [s]\ \mathsf{city\text{-}of}(loc) \not\doteq \mathsf{city\text{-}of}(loc')]$ is generated for the load-plane operator by the only-load-into-plane-when-necessary control rule: There must be a goal that the object *obj* to be loaded into the plane should be in another city. This formula only needs to evaluate fluents at time s, which is the invocation timepoint of load-plane.

Given this transformation, TALplanner has a chance of detecting that an action would cause a control rule violation even before the action is applied. This, in turn, reduces the number of actions applied by TALplanner and thereby also the number of states that must be generated (the right hand side of Figure 8.3).

Whether or not the generation of precondition control has a significant impact on the total performance of TALplanner depends on the characteristics of the planning domain and the control formulas. A greater performance improvement can naturally be found in domains where precondition control formulas are often not satisfied, since this is a prerequisite for being able to reduce the number of actions to be applied. The total performance improvements also depend on the relation between the time required to evaluate a control formula and the time required to apply an action. If states are simple and can be constructed quickly, actions are simple and do not have complex quantified or conditional effects, and the precondition control formulas that can be generated are complex, then applying actions may take only an insignificant part of the total time required by the planner. In this case, reducing the number of actions to be applied may have a negligible impact on the performance of the planner. Conversely, if states and actions are complex and precondition control formulas are simple, the performance impact of precondition control can be very significant.

But if a control rule can be expressed as a precondition, why not simply write it that way? In fact, the use of *manually* specified precondition control in TLPlan has been discussed independently by Bacchus and Ady (1999), and yielded similar improvements compared to the progression algorithm usually used by TLPlan. However, there are several reasons why the use of control rules is often better, perhaps the most important of which is that it allows a more modular specification of the control knowledge: Each constraint is specified as a single control rule, rather than as a number of (possibly different) preconditions in each operator. Allowing an automatic analyzer to generate preconditions wherever possible should also be less error-prone, especially for more complex rules where interdependencies between multiple actions must be taken into account. This is done by TALplanner.

8.7 Empirical Benchmark Tests

An earlier, considerably less general version of the domain analysis techniques presented in this chapter was implemented in the version of TALplanner that competed in the Second International Planning Competition (IPC-2000; see Chapter 9). This version was also tested using the same benchmark domains and problem instances already used in Section 6.7 on page 186. The results, initially presented in Kvarnström and Doherty (2000b), are mostly superceded by more current tests but are still interesting in two respects: They provide a snapshot of the performance that could be expected from state-of-the-art hand-tailored planners in early 2000, and they can be directly compared to the benchmarks from early 1999 that were presented in Chapter 6. The results are therefore reproduced in Section 8.7.1 below.

When the full repertoire of optimization techniques in this chapter had been developed and implemented, we performed a more thorough examination of the benefits of each type of optimization technique. The results are presented in Section 8.7.2 on page 233.

8.7.1 Initial Testing

Updated Results for the Logistics Domain. In our initial testing of domain analysis and formula optimization techniques, benchmark results were once more generated for TALplanner with TAL-based control rules using the 30 logistics problem from the First International Planning Competition (IPC-1998, McDermott, 1998). The results shown in Table 8.1 clearly demonstrate a great speedup for TALplanner when compared to the previous version of the planner.

As before, TLPlan could not solve two of the problems using 256 MB of memory; the remaining problems required between 0.4 seconds and 17 hours to complete. TALplanner proved to be considerably more efficient: The longest plan (for problem 29) contained 330 actions and was created in approximately 0.3 seconds, while the most complex problem (problem 28) resulted in 274 actions and required 0.63 seconds.

We also compared TALplanner to the most recent version of the HTN planner SHOP, using the standard SHOP formalization of the logistics domain. In Nau, Cau, Lotem, and Muños-Avila (1999), SHOP was found to be considerably faster than TLPlan. Nau et al. believed the most important reason to be the fact that SHOP in effect allows the user to design a planning algorithm, rather than prune a search space, and that SHOP's use of problem reduction can be more efficient than the state space search used by TLPlan. However, even though the SHOP results in Table 8.1 were generated using a newer, considerably faster version of SHOP than the one used in Nau et al. (1999), TALplanner is still faster by almost two orders of magnitude for some of the larger problem instances, despite running on a slower computer.

Updated Results for the Blocks World. For the blocks world, we used a set of test problems with between 25 and 5000 blocks, including the 14 problems previously used in Section 6.7. For TLPlan, the world definition and control rules from `domains/Blocks/4OpsBlocksWorld.tlp` in the TLPlan distribution were used together with an additional control rule ensuring that blocks are not placed on the table if their final destinations are ready. This additional rule resulted in shorter plans as well as improved performance. For TALplanner, the same control rules were translated into TAL.

Table 8.2 contains the results (times are in seconds). TLPlan was tested on the first ten problems; for the larger problems, 256 MB of memory was not sufficient. TALplanner solved the entire set of problem instances in less than one minute and required approximately 70 MB of memory for the largest problem instance.

Program Versions and Test Procedures. Test results for TLPlan and TALplanner were generated on a 333 MHz Pentium II computer running Windows NT 4.0 SP3, with 256 MB of memory. In the logistics domain, TALplanner was also compared with SHOP (Nau et al., 1999), a hierarchical task network (HTN) planner, which ran on a 440 MHz Sun Ultra 10 with 256 MB of memory.

The TLPlan tests were performed using the precompiled C version that can be downloaded from `http://www.cs.toronto.edu/~fbacchus/`. TALplanner is written in Java, and TALplanner 2.741 was used together with the Java Development Kit 1.2.2-001 and the HotSpot virtual machine (2.0rc2), both of which can be downloaded

				TALplanner	
	Ops	TLPlan	SHOP	before	now
1	26	0.421	0.060	0.160	0.040
2	33	1.712	0.090	0.501	0.040
3	55	19.398	0.210	3.505	0.060
4	59	54.338	0.250	7.581	0.060
5	22	0.310	0.060	0.250	0.030
6	72	84.191	0.480	19.919	0.070
7	34	5.568	0.120	1.032	0.041
8	41	97.310	0.360	14.241	0.061
9	85	218.644	0.420	24.405	0.080
10	105	167.581	1.260	18.436	0.080
11	31	5.167	0.100	0.872	0.040
12	41	286.021	0.760	41.840	0.050
13	67	1073.263	1.160	50.753	0.070
14	94	802.824	1.000	36.282	0.100
15	94	24.675	0.260	3.135	0.080
16	58	168.002	0.390	10.966	0.060
17	45	90.460	0.210	6.850	0.100
18	170	4358.367	2.690	104.009	0.120
19	153	2685.021	0.960	57.112	0.110
20	150	3414.089	1.420	141.984	0.121
21	104	2102.643	0.860	95.597	0.100
22	296		11.570	1027.097	0.330
23	115	116.798	0.810	9.954	0.080
24	41	695.780	0.320	47.158	0.070
25	190	11724.910	2.530	680.428	0.220
26	194	9976.946	19.200	699.576	0.230
27	149	14994.551	1.290	463.737	0.220
28	274		6.720	4613.333	0.631
29	330	60874.834	15.510	545.845	0.291
30	136	14070.923	3.860	1036.160	0.211
		Pentium II-333	UltraSparc-440	Pentium II-333	

Table 8.1: Test Results for the Logistics Domain

from http://java.sun.com. SHOP is written in Lisp, and SHOP 1.6.1 was used together with Allegro Common Lisp Enterprise Edition 5.0.

8.7.2 Benchmark Analysis

The complete optimizer is based on the optimizations specified above – but it is not possible to determine the total performance improvements given an analy-

		Plan length	TLPlan	TALplanner before	TALplanner now
16	25	16	0.100	0.060	0.090
24	50	68	1.843	1.302	0.090
25	70	86	5.528	3.715	0.090
26	70	104	7.060	4.406	0.110
27	100	158	35.321	14.060	0.110
28	140	230	175.893	41.940	0.130
29	200	350	734.546	142.996	0.151
30	280	350	2918.847	315.654	0.160
31	280	470	3067.261	474.012	0.190
32	460	470	20745.280	1899.992	0.210
33	460	794		2535.946	0.331
34	640	1118		7679.733	0.461
35	820	1478		12837.629	0.601
36	1000	1802		25028.509	0.771
37	1400	2450			1.111
38	1400	2630			1.192
39	2000	3278			1.733
40	2000	3710			1.922
41	5000	3710			6.920
42	5000	9326			8.702
43	5000	15314			18.327

Table 8.2: Test Results for the Blocks World

sis of each individual optimization. Instead, it is often the case that optimization techniques have no discernable performance impact at all when taken in isolation, but that combinations of such apparently irrelevant techniques can cooperate to provide a greater synergistic effect. Nevertheless, further benchmark testing and analysis of the current domain analysis algorithms shows that a certain pattern appears in most domains.

Operator-specific control rule analysis is absolutely essential to the performance of the planner, to the extent that removing it generally makes the generation of precondition control impossible (since this requires the reduction and removal of control rule disjuncts referring to the "future", which can only be done with an operator-specific analysis) and makes the use of state invariants ineffective.

When the operator-specific analysis is added, the generation of precondition control has a significant effect. Finally, when precondition control has been introduced, far fewer states are expanded and the speed of testing preconditions becomes more important to the performance of the planner. This makes the use of state invariants more significant, since they can be used to further simplify the augmented preconditions.

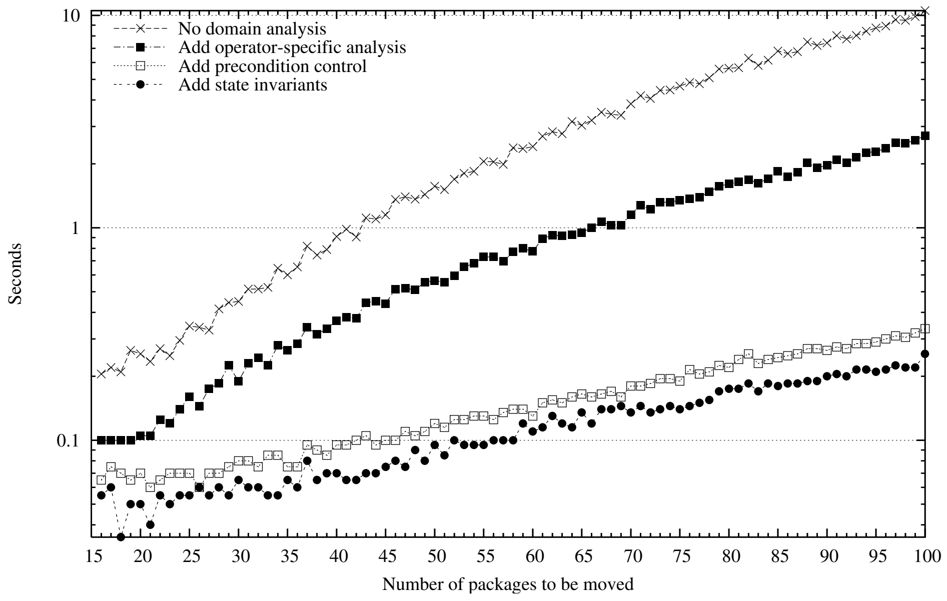


Figure 8.4: Control Analysis Results for IPC-2000 Logistics Problems

This is demonstrated in a set of benchmark tests using planning domains from the IPC-2000 and IPC-2002 planning competitions. The tests from IPC-2000 were run on an 800 MHz Pentium III machine with 512 MB of RAM, running Red Hat Linux 7.1 and Java 1.3. The tests from IPC-2002 were run on a 3000 MHz Pentium 4 machine with 2 GB of RAM, running Windows XP and Java 1.5.

Logistics. Competitors in the hand-tailored track of IPC-2000 were provided with a set of logistics problem instances containing between 16 and 100 packages to be moved. With two distinct problem variations for each problem size, there were a total of 170 problem instances to be solved. Figure 8.4 shows the average time required for TALplanner to generate plans for each problem size given four different levels of domain analysis. The topmost line indicates the time required to generate plans without the new domain analysis techniques. Adding operator-specific analysis improves performance by a factor of up to 4 for the largest problem instances. Adding precondition control yields another factor of 8, and finally, adding state invariants reduces the amount of time used by a factor of 1.3. The accumulated performance improvement for all domain analysis techniques ranges from approximately a factor of 3 for the smallest logistics instances to a factor of over 40 for the largest problem instances.

Blocks World. The blocks world was also used in IPC-2000 (Figure 8.5). Here, operator-specific analysis results in an 8-fold speedup for the largest problem instances with 500 blocks, after which adding precondition control results reduces

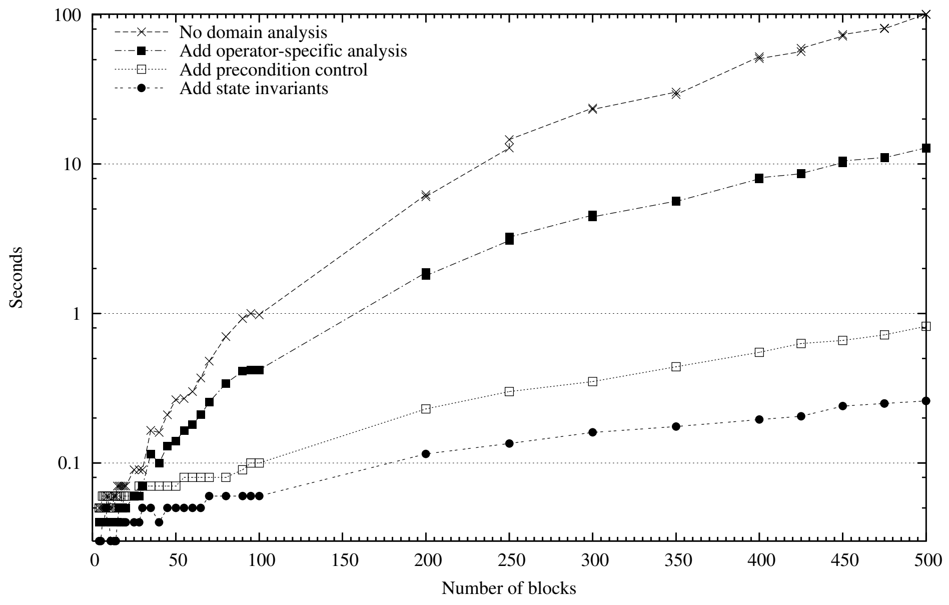


Figure 8.5: Control Analysis Results for IPC-2000 Blocks World Problems

the time by a factor of 16 and the use of state invariants yields another factor of 3. In total, these domain analysis techniques make TALplanner up to 400 times faster for the largest problem instances. It should be noted that the improvements caused by domain analysis are partly due to the elimination of quantifiers and therefore do not result in a constant factor speedup but a reduction in time complexity for certain operations. Thus, larger problem instances are affected to a greater degree.

Depots. The Depots domain, which will be described in further detail in Section 9.2.3 on page 262, was part of the IPC-2002 planning competition. The test results presented in Figure 8.6 were generated using the 22 larger problem instances intended for the "handcoded" track of the competition, and plans contain approximately 100 to 800 operator instances. In this domain, operator-specific analysis appears to improve performance by a factor of 3 to 5 for the larger problem instances. Precondition control improves performance by another factor of 5 to 10. The performance impact of adding state invariants appears to be negligible for problem instances of this size. In total, applying these domain analysis techniques to the Depots domain generally allows TALplanner to generate plans between 10 and 100 times faster than without domain analysis.

Rovers. Finally we consider the Rovers domain, which will be described in further detail in Section 9.2.5 on page 268. Like the Depots domain, the Rovers domain was part of the IPC-2002 planning competition, and the test results presented in Figure 8.7 were generated using the 20 larger problem instances intended for the

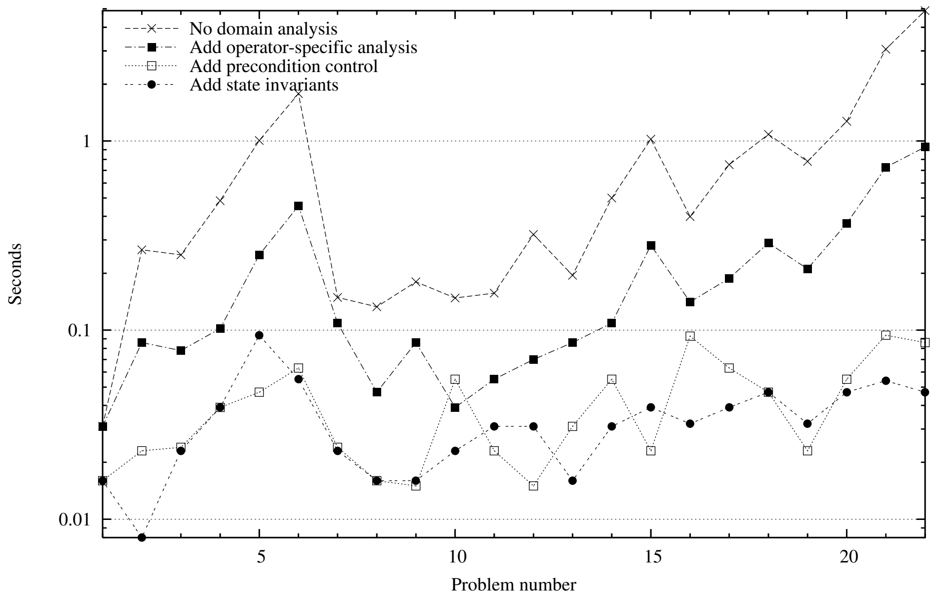


Figure 8.6: Control Analysis Results for IPC-2002 Depots STRIPS Problems

"handcoded" track of the competition. For this domain, plans consist of approximately 35 to 200 actions.

The Rovers domain provides a counterpoint to the three preceding domains. Problem instances are rather small, and the branching factor is not large: The largest problem instances do not cause TALplanner to investigate more than 2000 states, even with all optimizations turned off. In this situation, no significant improvements can be made. Adding operator-specific analysis does improve planning time by a factor of up to 10 for many largest problem instances, but precondition control yields no further measurable improvements. The TALplanner domain definition for the Rovers domain does not use state invariants.

8.8 Related Work

The algorithms developed in this chapter are intended to be used in a general formula optimizer, which should be applied to any formula, whether control-related or not, without necessitating an artificial increase in the number of operators or fluents in the domain and most importantly without affecting the set of possible plans and without restricting the expressivity of operators or control formulas. If these restrictions are lifted, a number of new possibilities open up, especially in the area of generating precondition-based control from standard temporal control formulas.

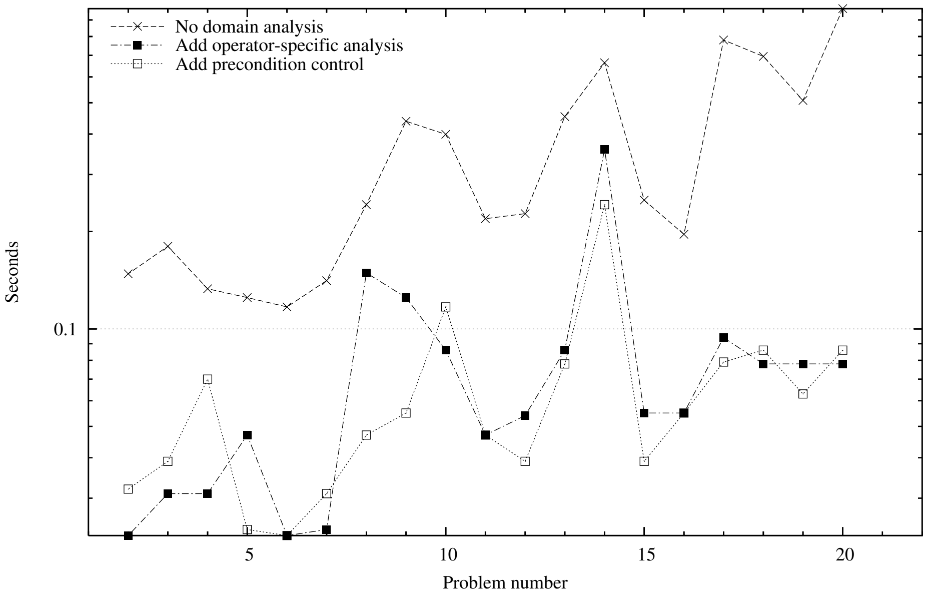


Figure 8.7: Control Analysis Results for IPC-2002 Rovers STRIPS Problems

Rintanen (2000a) presents a method for compiling an LTL control formula and a set of operators into a new set of operators with explicitly embedded control knowledge. Unlike TALplanner's generation of precondition control, this method can be applied to various forms of the "until" (U) operator. However, the method is still limited to control rules having certain simple syntactic forms. It also alters the planning domain by adding new facts keeping track of the current control state, and operators are altered by adding new effects updating those facts.

Cresswell and Coddington (2004) describe a method for compiling LTL formulas into PDDL, based on the generation of a finite state machine corresponding to an arbitrary LTL formula. The current state of the finite state machine is modeled using a new fsm_state fluent, which is updated using automatically generated conditional effects. As described by the authors, these updates may force a total order between actions that could otherwise have been placed in parallel.

Chapter 9

Planning Competitions

The first International Planning Competition was held in 1998, at the Fourth International Conference of Artificial Intelligence Planning and Scheduling (AIPS-98). Intended to provide empirical comparisons of existing planning systems as well as an incitement for improving planners and extending their applicability, the competition attracted five different planning systems: Blackbox (Kautz & Selman, 1998, 1999), HSP (Bonet & Geffner, 1998), Sensory Graphplan (Anderson, Smith, & Weld, 1998; Weld, Anderson, & Smith, 1998), STAN (Long & Fox, 1999) and IPP (Koehler et al., 1997). Though the results were somewhat inconclusive in the sense that no planner consistently outperformed the others, the competition was still a success, and new competitions have been held every other year.

Unlike the first competition, the second International Planning Competition (IPC-2 or IPC-2000) had a track for hand-tailored or knowledge-based planners. TALplanner participated in this competition as well as in IPC-3 (IPC-2002). This chapter presents the results from these competitions together with an extensive discussion of the planning domains from IPC-2002 and how control rules were developed for these domains.

While reading this chapter, please keep in mind that knowledge-based planners cannot easily be tested in isolation. Whereas all fully automated planners used identical domain definitions – the operators and facts specified by the competition organizers – each team working with a knowledge-based planner had to generate their own additional domain knowledge to be used with their particular planner. What is truly tested is the combination of planner and domain knowledge. The effort spent on investigating the properties of each domain and encoding the appropriate domain knowledge may vary between teams, which can easily explain small constant factor differences in performance as well as plan quality. Some differences may even be explained by one team having some luck when randomly choosing one control approach over another.

9.1 International Planning Competition 2000

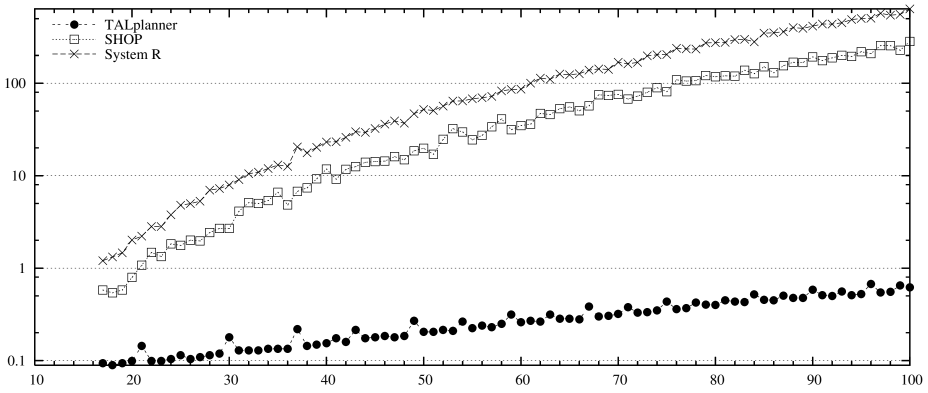
TALplanner participated in the hand-tailored planning track of the second International Planning Competition in 2000 (IPC-2000). The version that was used in the competition already contained an early version of the domain analysis improvements specified in Chapter 8, which helped TALplanner win the "distinguished planner" award, outperforming other hand-tailored planners by orders of magnitude in several planning domains.[1] TALplanner also won first place in the ADL-plus-resources track of the Miconic 10 elevator control domain competition (Koehler, 2000) sponsored by Schindler Lifts Ltd. and taking place as part of IPC-2000. The planning competition and the participating planners were later described in the fall 2001 issue of Artificial Intelligence Magazine (Bacchus, 2001; Doherty & Kvarnström, 2001).

In this section, we will briefly present the results from the hand-tailored planning track. These benchmark results were generated on a 500 MHz Pentium III machine with 1 GB of memory provided by the competition organizers. Results will be presented for all five domains used in the hand-tailored track of the competition: The blocks world, the logistics domain, the schedule domain, the Freecell domain, and the Miconic 10 elevator domain. The complete PDDL domain definitions and problem instances are available from the IPC-2000 home page[2], together with the raw result data files from which the graphs in this section were created.

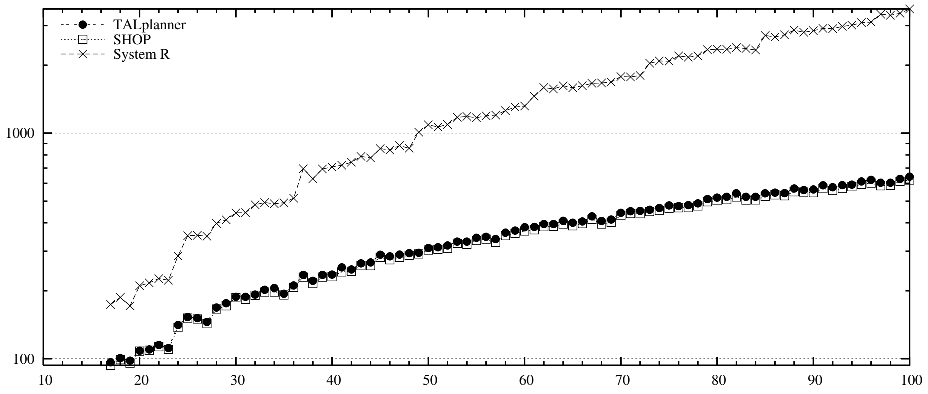
The results for TALplanner are compared with those from several other planners. SHOP (Nau et al., 1999, 2001) is a hierarchical task network (HTN) planner, which does not attempt to achieve a goal but to perform a task. Tasks can be decomposed into subtasks according to domain-specific rules, until all remaining tasks are primitive tasks, corresponding to planning operators. System R (Lin, 2001) is a regression-based STRIPS-like planner where domain-dependent control information is used to order subgoals, prune subgoals, and determine the way a subgoal is solved by regressing it to a new conjunctive goal. PbR (Ambite, 1998; Ambite et al., 2000; Ambite & Knoblock, 2001), Planning by Rewriting, begins by quickly generating a potentially inefficient plan and then applies domain-specific rewriting rules to gradually improve the result. BDDPlan (Störr, 2001) uses Binary Decision Diagrams to support reasoning in the Fluent Calculus, where a model checking algorithm is used to do an implicit breadth first search. PropPlan (Fourman, 2000) also uses Binary Decision Diagrams. Some of these planners did not participate in all domains.

[1]It should be noted that TLPlan did not participate in this track, since one of its authors arranged the competition.

[2]`http://www.cs.toronto.edu/aips2000/`



(a) Average time (seconds) depending on number of packages



(b) Average plan length depending on number of packages

Figure 9.1: Results for IPC-2000 Logistics Problems

9.1.1 The Logistics Domain

The first domain used in the second International Planning Competition was the standard logistics domain with the standard six ADL-style operators. The logistics domain has already been used as an example earlier in this thesis. The control knowledge used in the competition was based on an adapted version of the control formulas used by Bacchus and Kabanza (Bacchus & Kabanza, 2000) with several additions and improvements to decrease plan length and time requirements.

Figure 9.1 contains the logistics results from the hand-tailored track of IPC-2000. The x axis indicates the number of packages in each problem instance, ranging from 17 to 100. There were two problem instances for each problem size; results have been averaged for all instances having the same number of packages. Note the small spikes in the curve for TALplanner, resulting from Java garbage collection.

As can be seen in the figure, SHOP outperformed System R by slightly more than a factor of two. The relative difference between the two systems remained approximately constant over a large set of problem instances, indicating that the time complexity of the two planners was essentially the same for the logistics domain. TALplanner, on the other hand, increases its lead over the other two planners as problem sizes grow, ending up faster than SHOP by a factor of over 400 for the largest problem instances. In terms of plan quality, TALplanner and SHOP generate plans containing approximately the same number of actions, while the plans generated by System R are up to six times as long.

9.1.2 The Blocks World

The second domain used in IPC-2000 was the standard blocks world with four ADL-style operators. Again, this domain has already been used as an example in this thesis, and control knowledge was based on the control knowledge from Bacchus and Kabanza with some improvements that generate shorter plans under certain circumstances.

Figure 9.2 shows the blocks world results from the hand-tailored track of IPC-2000, where the x axis indicates the number of blocks for each problem instance. As in the logistics domain, there were two distincs problem instances for each problem size, and all results have been averaged for all instances of the same size.

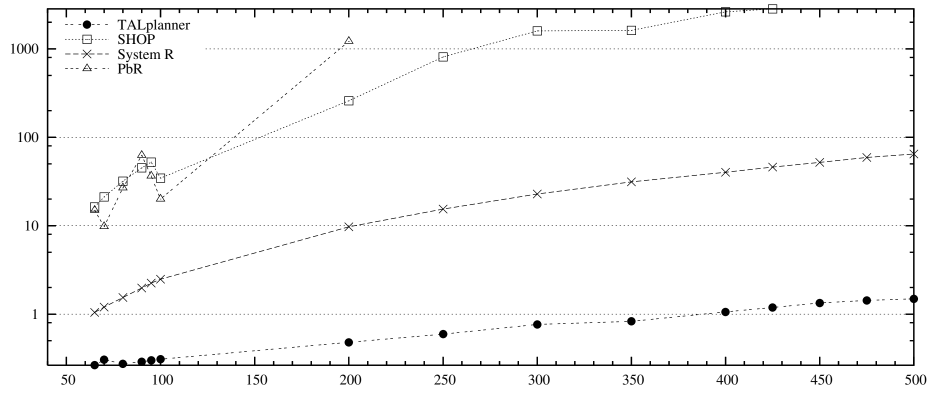
Once more, TALplanner outperformed the competing planners by a significant margin, taking around two seconds to generate plans for 500 blocks. System R required up to 65 seconds, and this time generated plans only 10% longer than those of the competing planners. SHOP used up to 4000 seconds and did not generate plans for the largest problem instances, while the behavior of PbR was somewhat uneven.

9.1.3 The Freecell Domain

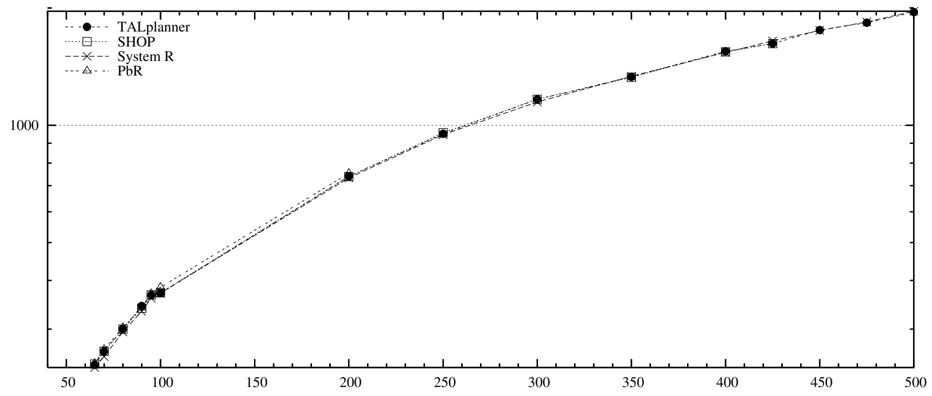
Freecell is a solitaire card game where 52 cards are dealt into 8 columns, each card visible to the player but only the last card of each column fully exposed. There are also four *foundations*, empty locations where each suit should eventually be placed in order of rank, and four *freecells*, each of which can temporarily hold a single card in order to allow the player some more freedom when moving cards around. An exposed card can be moved from a column or freecell to a column if it is placed on top of a card of the next higher rank and a different color (black 8 on red 9), from a column or freecell to a foundation in order of rank (one foundation per suit), or from a column to any empty freecell.

There were 60 problem instances of increasing complexity for the FreeCell domain in IPC-2000. Results for TALplanner, SHOP and System R are shown in Figure 9.3.

As can be seen in the figure, TALplanner solved all problems, which no other hand-tailored planner did. In terms of plan quality, TALplanner often produces



(a) Average time (seconds) depending on number of blocks

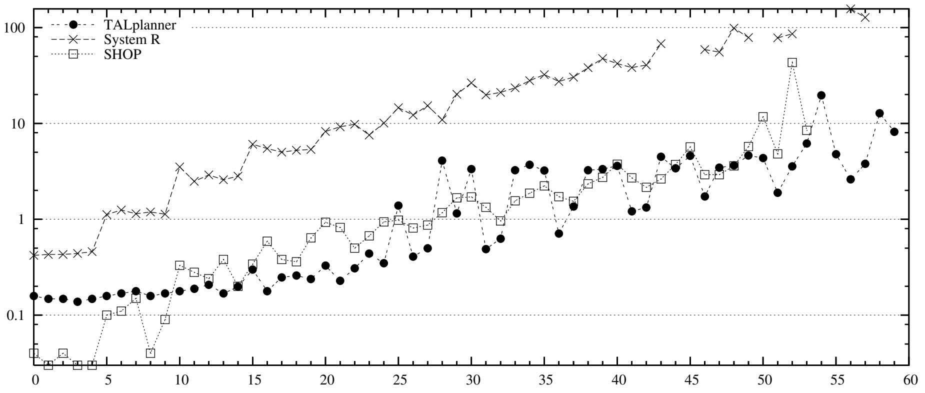


(b) Average plan length depending on number of blocks

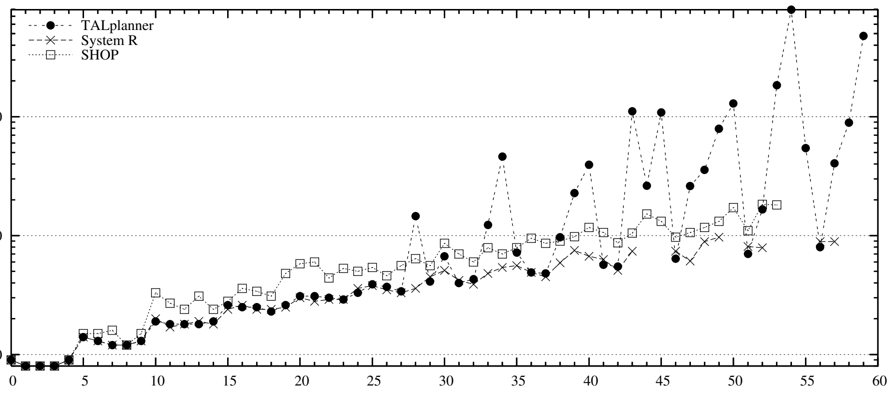
Figure 9.2: Results for IPC-2000 Blocks Problems

plans that are better than those generated by SHOP and equally good to those generatedy by System R – but sometimes, it generates plans that are several times longer. The most extreme example is number 44, "freecell-10-4", where TALplanner requires 1110 operators as opposed to 74 operators for System R and 105 operators for SHOP.

Why should TALplanner generate longer plans? As always, the performance of a hand-tailored planner depends to a great degree on the domain knowledge of the person writing the domain specification. Because we had never before played Freecell, it was unusually difficult to find reasonable rules that allowed us to purge moves that were "definitely stupid". After all, winning strategies in card games and board games often involve taking "counter-intuitive" losses that later allow you to regain what was lost, and finding these strategies requires some experience and learning on the part of the player. Better domain knowledge would definitely have improved the plans generated by TALplanner.



(a) Timing (Seconds)



(b) Plan Length

Figure 9.3: Results for IPC-2000 Freecell Problems

This domain also serves to highlight the potential conflict between generating good plans and generating plans quickly, and the difficulty in choosing a good balance between these two performance criteria when generating a domain specification for an abstract competition rather than having a concrete use in mind. Interestingly, removing certain control rules from the TALplanner domain specification allowed TALplanner to generate plans considerably more quickly – but the plans were considerably longer. Similarly, we could allow the planner to spend more time on plan search instead of being satisfied with the very first plan found in the pruned search tree. Whether the improvements in plan quality would be worth the additional time can only be determined in relation to a specific use, and for the planning competition we guessed that time requirements would be considered more important than plan lengths.

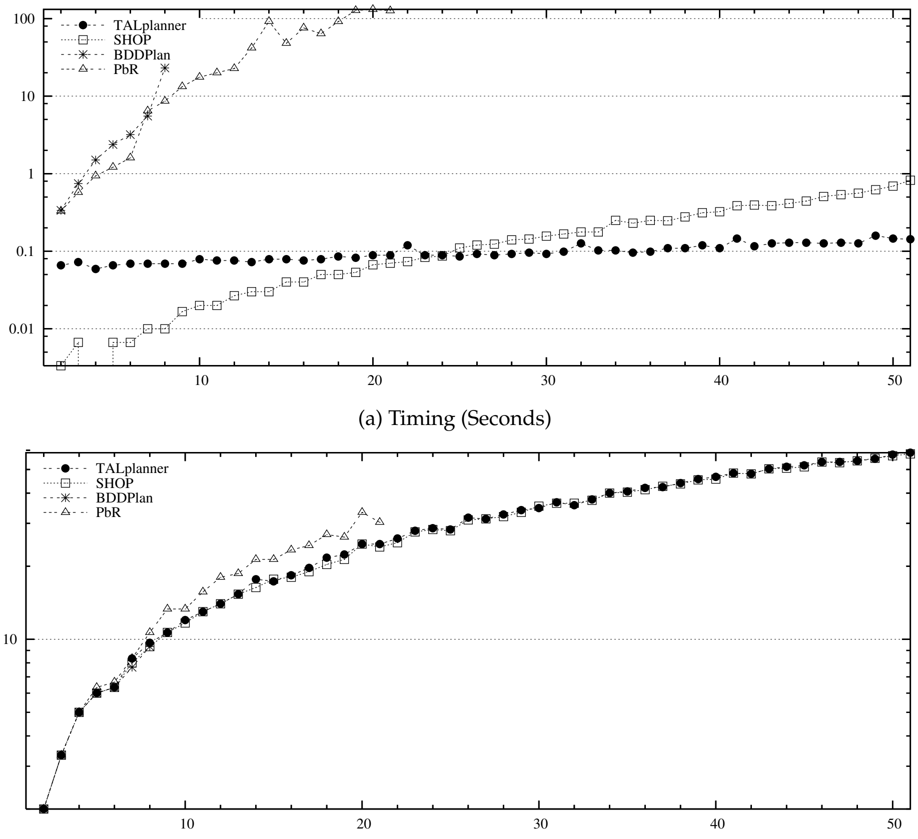


Figure 9.4: Results for IPC-2000 Schedule Problems

9.1.4 The Schedule Domain

In the schedule domain, there is a collection of parts and a number of operators that operate on these parts: polish, roll, lathe, grind, punch, drill-press, spray-paint, and immersion-paint. Each operator has a number of effects, some of which may undo the effects of other operators; for example, if a part has been painted, lathing it will have the side effect of removing the paint. The goal is for each part to have a certain shape, surface condition, and/or color.

A variety of problem sizes were made available, with between 2 and 51 parts to be scheduled. There were three problem instances for each problem size. The test results, which have been averaged for all instances of the same size, are shown in Figure 9.4. The number of parts to be scheduled increases along the x axis.

Here, TALplanner is slower than SHOP for the smallest problem instances. The reason for this is mainly that the startup time for the Java Virtual Machine and the

Just-In-Time compilation of TALplanner (4–6 seconds) has been distributed evenly over all problem instances. Due to the relatively small number of instances, this contributes almost a tenth of a second to each instance, which is significant due to the small size of each instance. For larger problems, TALplanner generally stays below 0.2 seconds, while SHOP requires up to one second. BDDPlan and PbR both require considerably more time and do not solve all problem instances.

In terms of quality, TALplanner, SHOP and BDDPlan consistently return plans of almost identical length, never differing by more than one or possibly two actions, while PbR generates somewhat longer plans.

9.1.5 The Miconic-10 Elevator Domain

Miconic 10 is an elevator system built by the Schindler Group.[3] The system is intended to be used in large buildings with multiple elevators, and is built on the idea of allowing passengers to key in their destination before an elevator actually arrives at their floor. This permits the system to place passengers headed for the same floor into the same elevator, minimizing the number of stops.

The problem faced by the central elevator controller can be expressed as a planning problem – and in fact, the commercial Miconic 10 product is based on AI planning techniques (Koehler & Schuster, 2000; Koehler, 2001; Koehler & Ottiger, 2002). Jana Koehler provided a set of Miconic 10 planning problems adapted to the PDDL planning domain definition language used in the competition (Koehler, 2000). The TALplanner results in this domain were somewhat erratic (Figure 9.5), often solving problems in well below a second but sometimes requiring up to 100 seconds, though the planner still outperformed the other hand-tailored planners. Better search control could probably ameliorate this problem. Another potential solution would be a modified search algorithm: Once depth first search chooses a "bad" action, it must explore all potential extensions to that plan (a complete subtree in the search tree) before being able to choose an alternative. Being able to temporarily skip back and try another choice of actions would probably improve search time in several difficult domains. This is a topic for future research.

9.2 International Planning Competition 2002

In the spring of 2002, TALplanner participated in the third International Planning Competition[4] (IPC-2002).

Compared to earlier years, this competition and its domain definition language PDDL2.1 (Fox & Long, 2003) had a very strong emphasis on increasing the complexity of the problem domains used as benchmark tests and the expressivity required

[3]http://www.us.schindler.com/SEC/websecen.nsf/pages/elev-MHR-Mic10-01

[4]http://www.dur.ac.uk/d.p.long/competition.html

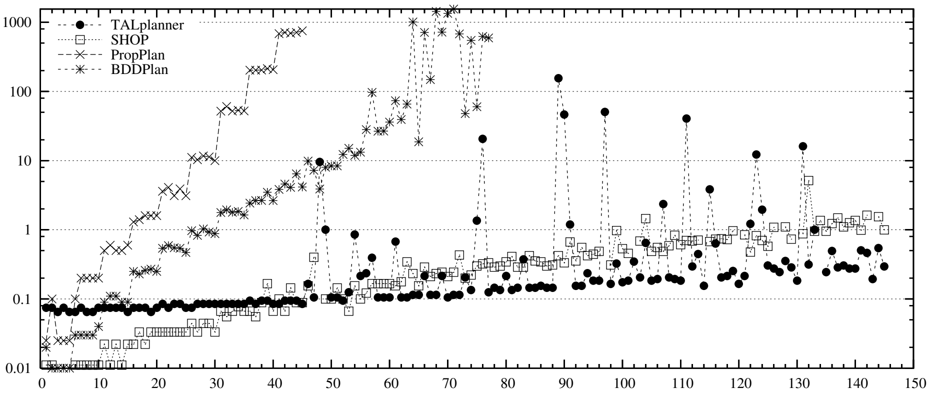


Figure 9.5: Timing Results for IPC-2000 Miconic-10 Elevator Domain Problems

to represent these domains in a planning system. Even in IPC-2000, two years before, all planning domains had mainly made use of STRIPS-level expressivity. Support for typed objects had not been required, and though some domains had used ADL-style quantified and conditional effects, restricted STRIPS versions of these domains had also been provided. In 2002, though, this was not sufficient. Multiple versions of each domain were provided to the competitors, many of which required support for conditional and quantified effects as well as operators with extended and context-dependent duration, non-integer time, and the use of numeric state variables. In order to generate plans of reasonable quality, support for concurrency was also an essential requirement in IPC-2002, whereas plan quality in IPC-2000 had been measured only in terms of the number of actions in a plan without regard to concurrency.

Although concurrent TALplanner had already been applied to a large number of domains, the competition provided us with a more varied set of domains that sometimes exploited concurrency in slightly different ways. This provided new ideas for improvements to TALplanner, and several minor enhancements to TALplanner's formula analysis algorithms were implemented during the first phase of the competition, allowing it to handle certain types of control formulas more efficiently when doing concurrent planning. These changes have been incorporated in the previous chapters of this thesis.

Of the eight planning domains in the third International Planning Competition, six were intended for hand-tailored planners. Except for the final domain, UMTranslog-2, all domains exist in at least four different variations: STRIPS, Numeric (where numeric quantities are involved), SimpleTime (where operators take constant non-unit time), and Timed (where operator durations may depend on the actual parameters in a specific operator invocation). TALplanner participated in all six domains, but due to lack of time for creating control rules, we limited our participation to the STRIPS, SimpleTime, and Timed versions of the domains.

In this section, which is mostly based on Kvarnström and Magnusson (2003), we will describe how the domains were translated from PDDL2.1 to TALplanner, and discuss some of the control rules that were created to handle the domains more efficiently. The main focus will be on two domains: ZenoTravel and Satellite. For these domains we will describe most of the control rules that were used in the competition as well as the incremental process of creating the rules, omitting only a few technical details and a couple of complex rules that turned out to have minimal impact on planner performance and plan quality. For the remaining domains (Depots, DriverLog, Rovers, and UMTranslog-2) we will describe the general intuitions behind our control rules, omitting the actual formulas.

We will also compare the performance of TALplanner with that of the other participants in the hand-tailored track: SHOP2 (Nau et al., 2003), TLPlan (Bacchus & Kabanza, 2000), and in some cases also FF (Hoffmann & Nebel, 2001), which is a fully automated planner but still participated in the hand-tailored track for two of the competition domains. All graphs are generated from the official collection of test results[5] that can be downloaded from the competition web page[6]. The time required to generate a plan is measured in seconds. Plan quality for STRIPS problem instances is measured using the number of actions in each generated plan. Plan quality for SimpleTime and Timed problem instances is measured in terms of the plan quality metric specified in each problem instance. In all domain variations except one, this metric was defined as the makespan of the generated plan, that is, the total time that would be required to execute this plan. As will be seen in Section 9.2.2, Timed ZenoTravel problem instances used another plan quality metric.

Further details regarding the basic setup of the competition, the planning domains being used, and timing and plan quality results, are available in Long and Fox (2003).

Before presenting control rules and benchmark results, we will begin with a few comments on the process of formalizing planning domains.

9.2.1 Pre-Defined Domains: Half the Work in Twice the Time?

In order to create a formal description of a real-world planning domain, it is of course always necessary to have a thorough understanding both of the domain itself and of how plans for the domain are eventually going to be used. There are several reasons why this is required, and most of these reasons are equally valid regardless of whether the formalization will eventually be used as the input to a fully automated planner or to a hand-tailored planner like TALplanner.

First, understanding the domain is required in order to determine what aspects of the domain truly need to be modeled (as types, predicates and functions) and what aspects can be abstracted away. For example, the standard formalization of

[5]http://planning.cis.strath.ac.uk/competition/IPCResults.tgz
[6]http://planning.cis.strath.ac.uk/competition/

the logistics domain does not model distances between locations, but allows trucks to move between any two locations in one time step. This is sufficient for some purposes, but a plan that is optimal given this abstraction may be extremely suboptimal if actually carried out by real trucks, which usually lack teleportation abilities. Similarly, it does not model package sizes or weights, or cargo capacities for trucks or airplanes. Neither does it model truck drivers, acceptable working hours for drivers, the additional costs incurred by overtime pay, or time required for maintenance activities such as changing to winter tires once a year. Which of these aspects need to be modeled depends very much on the particular application one has in mind.

Second, a detailed understanding of the domain is required in order to determine what operators are available to the planner and exactly how their preconditions and effects should be represented within the abstract logical model of the domain.

And finally, for hand-tailored planners, the domain must be understood in order to be able to guide a search algorithm using domain-dependent heuristics or control rules.

Usually all of these aspects of a domain are modeled at the same time, and much of the information and knowledge about the domain that was gathered in order to find a suitable set of predicates and operators – which is needed even for a fully automated planner – can be reused in the development of control rules or heuristics for a hand-tailored planner.

In the planning competition, however, the task is divided into two parts: The organizers define a set of domains using PDDL2.1, and then it is up to the competitors in the hand-tailored track to find suitable ways of guiding their planners. In one way, one could say that the competitors only need to do half the work, since the formalization is already done and only the task of finding control rules remains. Unfortunately it is still necessary to understand the domain just as thoroughly in order to write control rules. For the more complex domains, doing this half of the work in isolation might easily take twice the time, since all the constraints involved in the domain have to be understood from a PDDL2.1 formalization rather than by talking to domain experts. This is especially true for the complex UMTranslog-2 logistics domain, where a significant amount of time was spent trying to determine exactly how packages were allowed to move and how they can be loaded into and unloaded from various kinds of vehicle.

Another common problem occurring in most domains is that of striking a balance between quick search and high quality plans. Control rules can be written for either of these purposes, but they are no panacea – finding very high quality plans very quickly may be impossible, or the effort required to find the right rules may be prohibitive. Usually the only realistic option is to be satisfied with finding reasonable plans quickly or letting the planner spend some more time searching for plans of very high quality. Which option should be chosen depends on the application. In the planning competition, however, the results would not be applied in the usual sense of the word. Instead, we had to choose an approach based on how we expected the results to be judged in the competition. Consequently we aimed at the first option: Quickly producing plans of reasonable quality, often by forcing the planner to make a choice that is usually better even though it may occasionally be worse. Except in the satellite domain, where slew times unexpectedly did not satisfy the triangle inequality, we believe we succeeded fairly well.

Yet another problem caused by having to use a predefined formalization of a planning domain is that the degree of detail used in the model is determined in advance. In the real world there would more likely be a *minimum* level of detail required, and anything above this level would be acceptable. It may not seem like this should be a problem – intuitively, adding new details to a planning problem ought to make it harder, and so it would be best to remain at the minimum level of detail. But this is not always true, especially not when control rules are involved. This will be seen in the timed ZenoTravel domain, for example, where some control rules would be both simpler and more effective if it was possible to refuel to a specific level, just like in the real world, rather than just having a simple abstract refuel operator that unconditionally fills the tank completely.

This should not be taken as a complaint against the organization of the competition – allowing different planners to use different formalizations would of course be infeasible. Nevertheless, it does present some additional problems that are not encountered to the same degree in real-world domains and that deserve to be mentioned here.

9.2.2 The ZenoTravel Domain

In the ZenoTravel domain, there are a number of aircraft that can fly people between cities. There are five operators available: Persons may board and debark from aircraft, and aircraft may fly, zoom (fly quickly, using more fuel), and refuel. There are no restrictions on how many people an aircraft can carry. Flying and zooming are equivalent except that zooming is generally faster and uses more fuel. Figure 9.6 shows an example problem, with arrows pointing out goal locations.

ZenoTravel: STRIPS

Below we show the operator definitions for the STRIPS version of the ZenoTravel domain. These operators have been more or less directly translated from the PDDL representation. The main difference is that the PDDL representation uses PDDL2.1 level 1, with single-step actions, which has a stricter concept of mutual exclusion than TALplanner does and automatically enforces certain invariants, such as the fact that an aircraft should not leave if a person is boarding, because the location of the aircraft is modified by fly and used in the precondition of board. The TAL-C semantics used by TALplanner is more similar to PDDL2.1 level 3 (with durative actions), where such invariant conditions must be stated explicitly. This is done using

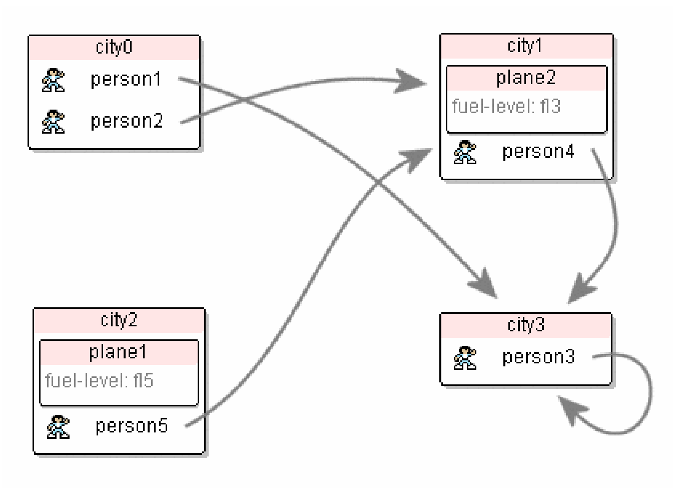


Figure 9.6: A ZenoTravel problem instance (STRIPS problem 6)

prevail conditions, which are considered to be separate from true *pre*-conditions. Note that in the STRIPS formalization fly and zoom take the same amount of time, since only single-step actions are possible.

```
operator   board($person, aircraft, city$) :at $s$
  :precond [$s$] at($person, city$) $\wedge$ at($aircraft, city$)
  :prevail [$s$+1] at($aircraft, city$)
  :effects [$s$+1] at($person, city$) := false, [$s$+1] in($person, aircraft$) := true

operator   debark($person, aircraft, city$) :at $s$
  :precond [$s$] in($person, aircraft$) $\wedge$ at($aircraft, city$)
  :prevail [$s$+1] at($aircraft, city$)
  :effects [$s$+1] in($person, aircraft$) := false, [$s$+1] at($person, city$) := true

operator   fly($aircraft, city_1, city_2, flevel_1, flevel_2$) :at $s$
  :precond [$s$] at($aircraft, city_1$) $\wedge$ fuel-level($aircraft, flevel_1$) $\wedge$ next($flevel_2, flevel_1$)
  :effects [$s$+1] at($aircraft, city_1$) := false, [$s$+1] fuel-level($aircraft, flevel_1$) := false,
           [$s$+1] at($aircraft, city_2$) := true,  [$s$+1] fuel-level($aircraft, flevel_2$) := true

operator   zoom($aircraft, city_1, city_2, flevel_1, flevel_2, flevel_3$) :at $s$
  :precond [$s$] at($aircraft, city_1$) $\wedge$ fuel-level($aircraft, flevel_1$) $\wedge$
  :nothing    next($flevel_2, flevel_1$) $\wedge$ next($flevel_3, flevel_2$)
  :effects [$s$+1] at($aircraft, city_1$) := false, [$s$+1] fuel-level($aircraft, flevel_1$) := false,
           [$s$+1] at($aircraft, city_2$) := true,  [$s$+1] fuel-level($aircraft, flevel_3$) := true

operator   refuel($aircraft, city, flevel, flevel_1$) :at $s$
  :precond [$s$] fuel-level($aircraft, flevel$) $\wedge$ next($flevel, flevel_1$) $\wedge$ at($aircraft, city$)
  :prevail [$s$+1] at($aircraft, city$)
  :effects [$s$+1] fuel-level($aircraft, flevel$) := false, [$s$+1] fuel-level($aircraft, flevel_1$) := true
```

After translating the operator definitions, it is time to create a set of control rules. There are basically two ways of doing this: First, one can sit down and think about suitable properties for a plan, and then write control rules that ensure that these properties will hold. Second, one can instruct the planner to show each branch that is explored in the search tree, and by observing the output one can identify "obviously stupid" choices made by the planner, such as choosing an action that inevitably leads to backtracking or performing actions that are useless given the goals. Control rules can then be written to prevent these branches of the tree from being explored. Both of these approaches will be covered here.

We begin with the first method, attempting to find a number of reasonable control rules simply by thinking about the properties of the ZenoTravel domain. Given some experience from other planning domains, this is in fact quite easy. For example, in many domains there are certain goals such that once they are satisfied, one should never allow them to be destroyed. In the ZenoTravel domain, people who are at their destinations never need to board an aircraft, which gives rise to the following control rule:

control :name "only-board-when-necessary"
$[t]\ \neg\mathsf{in}(\mathit{person}, \mathit{aircraft}) \wedge [t{+}1]\ \mathsf{in}(\mathit{person}, \mathit{aircraft}) \rightarrow$
$\exists \mathit{city}, \mathit{city}_2\ [\ [t]\ \mathsf{at}(\mathit{person}, \mathit{city}) \wedge \mathsf{goal}(\mathsf{at}(\mathit{person}, \mathit{city}_2)) \wedge \mathit{city} \neq \mathit{city}_2\ ]$

This TAL formula states that if we have a state transition from the person not being in the aircraft at time t to the person being in the aircraft at time $t + 1$, (that is, if the person just boarded the aircraft), then there must be a reason why this is allowed: It must be the case that the person is in a certain city and that there is a goal that the person should be in another city.

As noted previously control formulas can usually be written in many different forms. For example, it would have been equally valid to state that if a person is at a city (and therefore not in an aircraft), and is not required to be somewhere else, then at the next timepoint that person should still not be on board an aircraft:

control :name "only-board-when-necessary"
$[t]\ \mathsf{at}(\mathit{person}, \mathit{city}) \wedge \neg\exists \mathit{city}_2\ [\ \mathsf{goal}(\mathsf{at}(\mathit{person}, \mathit{city}_2)) \wedge \mathit{city} \neq \mathit{city}_2\ ] \rightarrow$
$[t{+}1]\ \neg\mathsf{in}(\mathit{person}, \mathit{aircraft})$

Note that although it may at first glance appear that a planner would have to be extraordinarily stupid to destroy goals that have already been satisfied, there are also many cases where temporarily destroying a goal is necessary in order to satisfy other goals. For example, if there is a goal that a certain aircraft should be at a certain location and it has already reached that destination, it might still have to fly a number of people to their destinations before it can return to its own destination.

Another natural idea (since aircraft do not follow predetermined routes in ZenoTravel, as they usually do in real life) would be to say that people should only debark when they have reached their final destination:

control :name "only-debark-when-in-goal-city"
$[t]\ \mathsf{in}(\mathit{person}, \mathit{aircraft}) \land [t{+}1]\ \neg\mathsf{in}(\mathit{person}, \mathit{aircraft}) \rightarrow$
$\exists \mathit{city}\ [\ [t]\ \mathsf{at}(\mathit{aircraft}, \mathit{city}) \land \mathsf{goal}(\mathsf{at}(\mathit{person}, \mathit{city}))\ ]$

There is a potential problem with this rule: In some cases an optimal plan might require a number of people to debark from one plane and then board a number of other planes, which could fly them to their destination concurrently, and this is strictly forbidden by only-debark-when-in-goal-city. This is a common problem that occurs for many planning domains, and it is up to the user to determine what to do depending on the requirements of the application for which the planner is being used.

There are a number of possible choices: We could ignore this problem and accept suboptimal plans, skip the rule completely and let the planner search through a vastly greater search space in order to find a plan which is guaranteed to be optimal, or as a compromise, attempt to create a weaker rule that does cut down the search space to some degree but gives optimal or closer-to-optimal plans. During the planning competition the conditions were somewhat artificial and were not clearly stated – would it be beneficial for a planner to spend ten times as much effort finding a plan if this plan was only five percent better, on average? We guessed that this would not be the case, and consequently we chose to include the control rule as stated above. In the future, a better solution would most likely be to *prefer* those plans where a person does not debark before reaching his destination but still *allow* other plans.

Given these two rules, we might now continue with the second approach to finding control rules. We run TALplanner on a simple problem instance and consider the operator sequences the planner examines during the depth first search process. This is the beginning of such a sequence for the problem instance in Figure 9.6. The complete plan generated by the planner contains 123 actions and requires 60 time steps. It is shown here in the IPC-2002 STRIPS result format where the timepoint at which an action is invoked is followed by the action itself.

```
0: (board person4 plane2 city1)        6: (fly plane1 city0 city1 fl2 fl1)
0: (board person5 plane1 city2)        6: (refuel plane2 city0 fl0 fl1)
1: (fly plane1 city2 city0 fl5 fl4)    7: (fly plane1 city1 city0 fl1 fl0)
1: (fly plane2 city1 city0 fl3 fl2)    7: (fly plane2 city0 city1 fl1 fl0)
2: (board person1 plane1 city0)        8: (refuel plane1 city0 fl0 fl1)
2: (board person2 plane2 city0)        8: (refuel plane2 city1 fl0 fl1)
3: (fly plane1 city0 city1 fl4 fl3)    9: (fly plane1 city0 city1 fl1 fl0)
3: (fly plane2 city0 city1 fl2 fl1)    9: (fly plane2 city1 city0 fl1 fl0)
4: (debark person2 plane2 city1)       10: (refuel plane1 city1 fl0 fl1)
4: (debark person5 plane1 city1)       11: (fly plane1 city1 city0 fl1 fl0)
5: (fly plane1 city1 city0 fl3 fl2)    11: (refuel plane2 city0 fl0 fl1)
5: (fly plane2 city1 city0 fl1 fl0)    ...
```

The beginning of the operator sequence appears to be reasonable, but after time 4, airplanes seem to be flying around randomly. There are no control rules guiding them, so apparently it was mainly luck that caused the planes to find reasonable cities to fly to at time 1 and 3. To make airplanes more goal-directed, we identify three important reasons why an airplane should move from *city* to $city_2$: that the goal asserts that the aircraft must end up in $city_2$when the plan is complete, that one of its passengers wants to go to $city_2$, or that there is a person waiting to be picked up by an airplane in $city_2$. The following rule formalizes these three intuitions:

```
control :name "planes-always-fly-to-goal"
$[t]\ \mathrm{at}(aircraft, city) \wedge [t{+}1]\ \neg\mathrm{at}(aircraft, city) \rightarrow$
$\exists city_2\ [\ [t{+}1]\ \mathrm{at}(aircraft, city_2) \wedge$
        $(\mathrm{goal}(\mathrm{at}(aircraft, city_2)) \vee$
         $\exists person\ [\ [t]\ \mathrm{in}(person, aircraft) \wedge \mathrm{goal}(\mathrm{at}(person, city_2))\ ] \vee$
         $\exists person\ [\ [t]\ \mathrm{at}(person, city_2) \wedge \mathrm{goal}(\neg\mathrm{at}(person, city_2))\ ])\ ]$
```

With these control rules, TALplanner can quickly produce a set of plans for the 20 "handcoded" problems from the IPC-2002 competition, and although the plans will not be optimal, they will not be nearly as bad as the example given above. Together, the plans require a total of 7164 actions and 618 time steps. The plan for the example in Figure 9.6 requires 20 actions and 7 time steps.

Nevertheless, there are still some improvements that can be made. The first criterion is too admissible: It allows a plane to visit its destination even if it still needs to pick up or drop off passengers. One way of preventing this would be to add the condition that all passengers must have reached their destinations:

```
define [t] all-persons-arrived:
$\forall person, city\ [\ \mathrm{goal}(\mathrm{at}(person, city)) \rightarrow [t]\ \mathrm{at}(person, city)\ ]$

control :name "planes-always-fly-to-goal"
$[t]\ \mathrm{at}(aircraft, city) \wedge [t{+}1]\ \neg\mathrm{at}(aircraft, city) \rightarrow$
$\exists city_2\ [\ [t{+}1]\ \mathrm{at}(aircraft, city_2) \wedge$
        $([t]\ \textbf{all-persons-arrived} \wedge \mathrm{goal}(\mathrm{at}(aircraft, city_2)) \vee$
         $\exists person\ [\ [t]\ \mathrm{in}(person, aircraft) \wedge \mathrm{goal}(\mathrm{at}(person, city_2))\ ] \vee$
         $\exists person\ [\ [t]\ \mathrm{at}(person, city_2) \wedge \mathrm{goal}(\neg\mathrm{at}(person, city_2))\ ])\ ]$
```

This improves plan quality slightly, and TALplanner now requires 7006 actions and 575 time steps. But the new control rule is in fact too strict, which can be seen in the following plan tail for handcoded STRIPS problem number 3:

```
14: (fly plane2 city4 city7 fl2 fl1)
14: (fly plane4 city8 city9 fl3 fl2)
14: (refuel plane1 city6 fl2 fl3)
14: (refuel plane3 city9 fl4 fl5)
15: (debark person24 plane4 city9)
15: (debark person28 plane4 city9)
15: (debark person34 plane2 city7)
15: (refuel plane1 city6 fl3 fl4)
15: (refuel plane2 city7 fl1 fl2)
15: (refuel plane3 city9 fl5 fl6)
15: (refuel plane4 city9 fl2 fl3)
16: (fly plane1 city6 city8 fl4 fl3)
16: (fly plane3 city9 city4 fl6 fl5)
```

In this example, **plane1** and **plane3** had to wait until all passengers had debarked from several other planes until they could go to their final destinations, even though we can clearly see that there was no real reason for them to wait, because all potential passengers had already been picked up and **plane1** and **plane3** already had enough fuel. We once again alter the control rule according to this new insight: A plane can go to its final destination if all passengers on board the plane are headed towards the same destination and there is no person left to be picked up (that is, all persons have already arrived or are currently on board planes).

```
define [t] all-persons-arrived-or-in-planes:
$\forall person, city$ [ goal(at($person, city$)) $\rightarrow$ [$t$] at($person, city$) $\vee$ $\exists aircraft$ [ in($person, aircraft$) ] ]

control :name "planes-always-fly-to-goal"
[$t$] at($aircraft, city$) $\wedge$ [$t$+1] $\neg$at($aircraft, city$) $\rightarrow$
$\exists city_2$ [ [$t$+1] at($aircraft, city_2$) $\wedge$
        ((goal(at($aircraft, city_2$)) $\wedge$ $\mathbf{[t]}$ all-persons-arrived-or-in-planes $\wedge$
        $\forall \mathbf{person}$ [ $\mathbf{[t]}$ in($\mathbf{person, aircraft}$) $\rightarrow$ goal(at($\mathbf{person, city_2}$)) ]) $\vee$
        $\exists person$ [ [$t$] in($person, aircraft$) $\wedge$ goal(at($person, city_2$)) ] $\vee$
        $\exists person$ [ [$t$] at($person, city_2$) $\wedge$ goal($\neg$at($person, city_2$)) ])]
```

This yields another minor improvement, and TALplanner now requires 6918 actions and 564 time steps. For the example used above, the end of the plan now looks as follows:

```
14: (fly plane1 city6 city8 fl2 fl1)        15: (debark person24 plane4 city9)
14: (fly plane2 city4 city7 fl2 fl1)        15: (debark person28 plane4 city9)
14: (fly plane4 city8 city9 fl3 fl2)        15: (debark person34 plane2 city7)
14: (refuel plane3 city9 fl4 fl5)           15: (fly plane3 city9 city4 fl5 fl4)
```

We once more study the plans generated by the current set of rules and quickly identify another obvious problem: Any number of airplanes may fly to the same location to pick up the same person. Once again, it is necessary to find a reasonable balance between finding optimal plans and finding plans quickly. In the contest, we attempted to find a high quality (but probably non-optimal) plan as quickly as possible. This was done by ensuring that no more than one airplane may go to any given place at the same time, *if* the sole purpose for going there is to pick up a person who is waiting:

```
control :name "planes-always-fly-to-goal"
[$t$] at($aircraft, city$) $\wedge$ [$t$+1] $\neg$at($aircraft, city$) $\rightarrow$
$\exists city_2$ [ [$t$+1] at($aircraft, city_2$) $\wedge$
        ((goal(at($aircraft, city_2$)) $\wedge$ [$t$] all-persons-arrived-or-in-planes $\wedge$
         $\forall person$ [ [$t$] in($person, aircraft$) $\rightarrow$ goal(at($person, city_2$)) ]) $\vee$
        $\exists person$ [ [$t$] in($person, aircraft$) $\wedge$ goal(at($person, city_2$)) ] $\vee$
        $\exists person$ [ [$t$] at($person, city_2$) $\wedge$ goal($\neg$at($person, city_2$)) ] $\wedge$
          $\neg\exists \mathbf{aircraft_2}$ [ $\mathbf{[t+1]}$ at($\mathbf{aircraft_2, city_2}$) $\wedge$ $\mathbf{aircraft_2 \neq aircraft}$ ])]
```

This rule provides a major improvement, and the complete set of plans now requires 5075 actions and 434 time steps.

So far, we have controlled where airplanes fly, when people board an airplane, and when they debark. There are no rules governing refueling, and a quick look at a plan for one of the larger problem instances reveals that whenever an aircraft has nothing else to do, it will refuel. This seems a little bit wasteful, but we are satisfied with adding a rule stating that airplanes must only refuel when their tanks are empty. This rule is not perfect, since an airplane may miss an opportunity to "pre-emptively" refuel and it can still refuel one fuel level even if it is not going to fly, but it does provide a significant improvement, bringing the number of actions down to 4234. The number of time steps is still 434.

A few minor adjustments were made to these rules before they were used in the competition. These adjustments include a modification to only-board-when-necessary to ensure that a person who must travel from *city* to $city_2$will choose a plane that already needs to visit both *city* and $city_2$, if this is possible, since this is less likely to increase the total number of flights.

One final change is prompted by the fact that the intended differences in timing between fly and zoom cannot be modeled correctly in the STRIPS version of the domain. Since all operators must take the same amount of time, the only difference between these two operators is that zoom uses twice as much fuel. Although it would have been possible to add a control rule ensuring that zoom was not used, it was easier to simply remove the zoom operator from the domain definition.

ZenoTravel: SimpleTime

The SimpleTime version of ZenoTravel is quite similar to the STRIPS version, the only difference being that actions may have non-unit duration and that certain preconditions must hold throughout the execution of an action. The TALplanner operator definitions are changed accordingly. For example, the board and fly operators can be changed as follows:

operator board(*person*, *aircraft*, *city*) :at *s*
  :precond [*s*] at(*person*, *city*) $\wedge$ at(*aircraft*, *city*)
  :prevail **[*s*+1, *s*+20]** at(*aircraft*, *city*)
  :**duration** **20**
  :effects [*s*+1] at(*person*, *city*) := **false**, **[*s*+20]** in(*person*, *aircraft*) := **true**

operator fly(*aircraft*, $city_1$, $city_2$, $flevel_1$, $flevel_2$) :at *s*
  :precond [*s*] at(*aircraft*, $city_1$) $\wedge$ fuel-level(*aircraft*, $flevel_1$) $\wedge$ next($flevel_2$, $flevel_1$)
  :**duration** **180**
  :effects [*s*+1] at(*aircraft*, $city_1$) := **false**, [*s*+1] fuel-level(*aircraft*, $flevel_1$) := **false**,
  [*s*+180] at(*aircraft*, $city_2$) := **true**, **[*s*+180]** fuel-level(*aircraft*, $flevel_2$) := **true**

If we run the planner on a set of SimpleTime problem instances, we get almost immediate results: The planner claims that there is no plan for any of the instances.

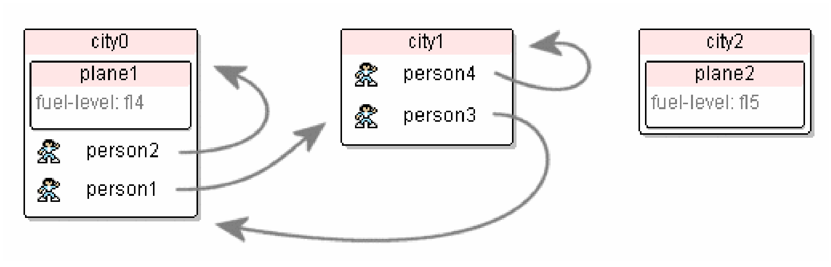


Figure 9.7: A ZenoTravel problem instance (SimpleTime problem 3)

The reason for this is, of course, that the control rules must be satisfied in any valid plan, and those rules were designed with the underlying assumption that actions had unit duration. For example, consider planes-always-fly-to-goal, which states that if a plane leaves a city at time t, it should be at a meaningful destination at t+1. When the fly action is invoked the plane must be at some city $city_1$, but beginning at the next time step there will be an interval where the aircraft is not present in any city at all, until it finally arrives in $city_2$ 180 time steps later. In other words, planes-always-fly-to-goal now ensures that the fly operator cannot be used at all, which is not quite what was originally intended.

One way of solving this problem would be to alter planes-always-fly-to-goal to say that if a plane leaves a city at time t, it should be at a meaningful destination at t+180. Unfortunately, the duration of the flight would then be encoded directly in the control rule instead of only in the operator, and so it would not work in the Timed version, where operators have variable durations – in fact, it would not even work in SimpleTime, because the zoom operator must also be taken into account.

Instead, the domain model is augmented with a new fluent flying-to(aircraft, city) which keeps track of whether a plane is flying, and if so, what its destination is. To ensure that this fluent is kept up-to-date, the following is added to the effects of the fly and zoom operators:

[s+1] flying-to($aircraft$, $city_2$) := **true**, [s+180] flying-to($aircraft$, $city_2$) := **false** // **for fly**
[s+1] flying-to($aircraft$, $city_2$) := **true**, [s+100] flying-to($aircraft$, $city_2$) := **false** // **for zoom**

The planes-always-fly-to-goal rule above can now be changed as follows, stating that if an aircraft ceases to be at $city$, then it must be flying to a reasonable destination:

control :name "planes-always-fly-to-goal"
[t] at($aircraft$, $city$) $\wedge$ [t+1] $\neg$at($aircraft$, $city$) $\rightarrow$
$\exists city_2$ [[t+1] **flying-to**($aircraft$, $city_2$) $\wedge \ldots$]

The same problem arises for boarding, and a new fluent boarding(`person`, `aircraft`) is added and used whenever necessary. Given these changes, the following are the first steps of the plan generated by TALplanner for the problem instance in Figure 9.7, shown in the IPC-2002 timed result format where the timepoint at which an action is invoked is followed by the action and the duration of the action:

```
0: (board person1 plane1 city0) [20]
20: (fly plane1 city0 city1 fl4 fl3) [180]
20: (zoom plane1 city0 city1 fl4 fl3 fl2) [100]
```

Intuitively, flying and zooming **plane1** at the same time should be impossible, but we have forgotten to specify this to the planner. Both actions have their preconditions satisfied at time 20, there are no prevail conditions, the initial effects at time 21 assign the flying-to fluent the same destination, and the final effects of the actions do not contradict each other since they take place at different timepoints: fly ends at time 200, while zoom ends at time 120.

There are several ways of specifying that fly and zoom are mutually exclusive. For example, it would be possible to introduce an interval effect stating that flying-to $(aircraft, city_2)$ must hold throughout the inner execution intervals of these actions, and become false at the end of each action:

[t+1,t+179] flying-to($aircraft$, $city_2$) := **true**, [t+180] flying-to($aircraft$, $city_2$) := **false** // **for fly**
[t+1,t+ 99] flying-to($aircraft$, $city_2$) := **true**, [t+100] flying-to($aircraft$, $city_2$) := **false** // **zoom**

It would also be possible to use a semaphore resource: An aircraft-specific resource with an initial value of 1, which can be borrowed exclusively by the fly and zoom actions. When one of these solutions is used, TALplanner finally rewards us with a short and correct plan:

```
0: (board person1 plane1 city0) [20]
20: (fly plane1 city0 city1 fl4 fl3) [180]
200: (board person3 plane1 city1) [20]
200: (debark person1 plane1 city1) [30]
230: (fly plane1 city1 city0 fl3 fl2) [180]
410: (debark person3 plane1 city0) [30]
;; Plan length 6, maxtime 440
```

Can it be improved? Remember that the STRIPS version never made use of the zoom operator. But in the SimpleTime version, flying takes 180 time steps and uses one unit of fuel, zooming takes 100 time steps and uses two units of fuel, and refueling one unit takes 73 time steps. $180 + 73$ is more than $100 + 2 \cdot 73$ and therefore we have the opposite situation: zoom is always better than fly. Commenting out the unwanted fly operator yields the following plan:

```
0: (board person1 plane1 city0) [20]
20: (zoom plane1 city0 city1 fl4 fl3 fl2) [100]
120: (board person3 plane1 city1) [20]
120: (debark person1 plane1 city1) [30]
150: (zoom plane1 city1 city0 fl2 fl1 fl0) [100]
250: (debark person3 plane1 city0) [30]
;; Plan length 6, maxtime 280
```

ZenoTravel: Timed

The Timed version further complicates the timing of the actions. Boarding and disembarking times are constant but problem-specific and are defined in the respective problem definition as two new functions, boarding-time and debarking-time. Refueling always fills the plane to its maximum capacity, but consumes time relative to the amount of fuel received and the refuel-rate of the aircraft. Each aircraft also has a fast-speed and a slow-speed with corresponding fast-burn and slow-burn fuel consumption. The distances between cities are specified using the distance($city_1$, $city_2$) function.

In the Timed version, operator durations have to be correctly calculated with a precision of three decimals, prompting a few minor changes to TALplanner. Once these changes had been implemented, few changes were needed to transform the SimpleTime domain to the Timed version.

The most important difference was perhaps the fact that depending on the speed and fuel consumption values defined in each problem and the situation where the operator is used, it is sometimes better to use the fly operator and sometimes better to use the zoom operator, unlike the STRIPS version where fly was always better and the SimpleTime domain where zoom was always better.

So when is zooming better than flying? It may seem like it would be easy to answer this question, given that we are only interested in minimizing time: Just check whether refueling the aircraft sufficiently to be able to zoom, followed by zooming to the destination, would be faster than only refueling enough to be able to fly and then flying more slowly to the destination. This is handled by the first clause in use-fly-instead-of-zoom below. The precondition of fly is then altered to require that use-fly-instead-of-zoom be true, and the precondition of zoom requires that use-fly-instead-of-zoom be false. If we had been interested in minimizing a combination of time and fuel usage, then this could also have been taken into account.

This is not quite sufficient to handle all problems, though. An airplane has a maximum fuel capacity, so if its destination is too distant, it may not be able to zoom. This is handled by the second clause in use-fly-instead-of-zoom.

Yet another problem is that it is not possible to tie one refueling action to each flight, as one would expect in the real world. There are two reasons for this problem.

First, airplanes may already have some fuel in the initial state, so in some situations a plane might zoom to its destination without incurring any additional cost, again assuming that the time required for executing the plan is the only metric being used – the plane already had enough fuel anyway and never had to refuel.

Second, unlike the SimpleTime version, an airplane cannot refuel "just enough" – the refuel operator always fills the tank completely. This change was most likely introduced in order to make the planning task easier by reducing the number of possible actions to choose from (for example, a planner that needs to create all ground instances of each operator might have some trouble if the refuel operator would take the amount of fuel as a floating point argument). But despite the probable intention behind this change, it introduces new problems for our control formulas. If a plane's tank is half full and this is enough fuel to zoom from A to B, it might then have to fill the *entire* tank before continuing to C, while if it used the fly operator, it might be able to continue to C without refueling at all. This means that one would have to take all possible future flights into account when determining whether to fly or zoom. If the domain had been modeled in more detail, this problem would not have existed.

Given these two complications, guaranteeing an optimal or near-optimal plan using a control rule is not easy, which is indeed only to be expected. For the competition we decided to be satisfied with a heuristic compromise, adding a third clause to use-fly-instead-of-zoom ensuring that if zooming would require refueling *immediately* but flying would not, the fly operator would be used.

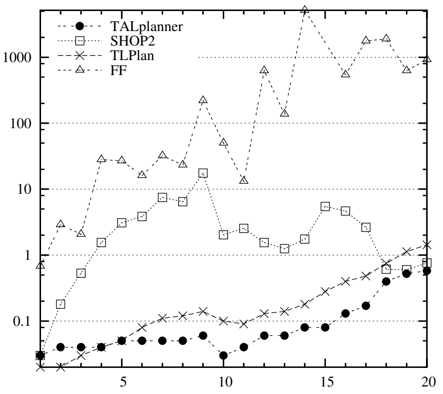
```
// Fly is (probably) better than zoom if:
define [t] use-fly-instead-of-zoom(aircraft, city_1, city_2):
  // If fly is faster wrt speed and refueling.
  ([t] (10000 / slow-speed(aircraft) + 10000 * slow-burn(aircraft) / refuel-rate(aircraft)) <
    (10000 / fast-speed(aircraft) +10000 * fast-burn(aircraft) / refuel-rate(aircraft))) ∨
  // If zoom is impossible across the given distance.
  ([t] distance(city_1, city_2) * fast-burn(aircraft) > capacity(aircraft)) ∨
  // If zoom has to refuel immediately but fly does not.
  ([t] fuel(aircraft) >= distance(city_1, city_2) * slow-burn(aircraft) ∧
    fuel(aircraft) < distance(city_1, city_2) * fast-burn(aircraft))
```

ZenoTravel: Discussion

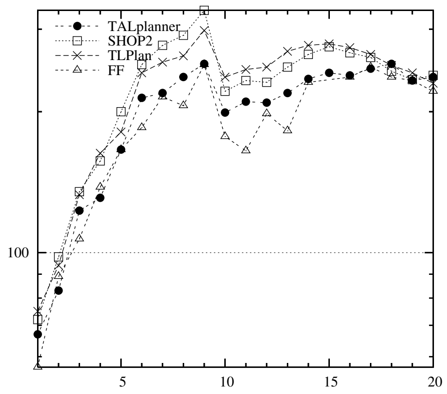
Finding control rules that yield good (but usually suboptimal) plans is not too difficult in the ZenoTravel domain. There are no risks involved in flying a plane to pick up passengers, since there are no limits on the number of passengers that can bord a single plane and since refueling is possible in any city. Also, since the graph of cities is fully connected, reasonable plans can generally be generated even without more advanced forms of route planning.

The benchmark results for this domain can be seen in Figure 9.8. In terms of performance, TALplanner is close to TLPlan: Somewhat better for STRIPS problems and the larger SimpleTime problems, and somewhat slower for the Timed problems. SHOP is generally considerably slower than either TALplanner or TLPlan. FF (Hoffmann & Nebel, 2001), which only participated in the STRIPS domain, is considerably slower than SHOP, though its performance is certainly very respectable considering its lack of domain-specific control knowledge.

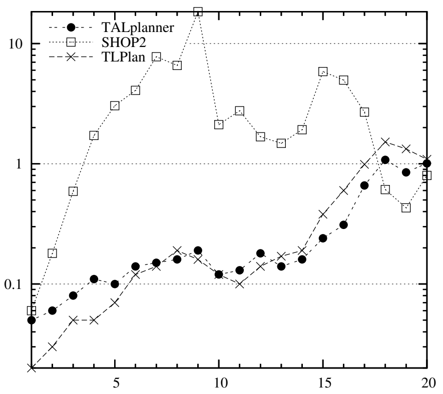
For the STRIPS version of ZenoTravel, plan quality is measured in terms of the number of actions in the solution plan. Here, TALplanner is more or less consistently somewhat better than TLPlan and SHOP2. FF often provides better or equally good plans, though given that it is a knowledge-sparse planner it also requires orders of magnitude more time to generate these plans. On the other hand,



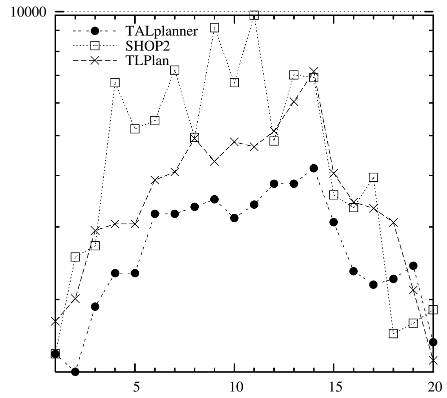
(a) Time (seconds, STRIPS)



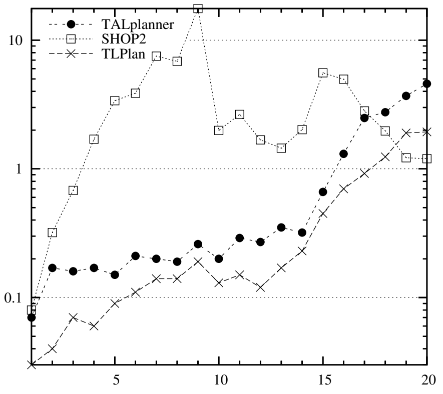
(b) Plan steps (STRIPS)



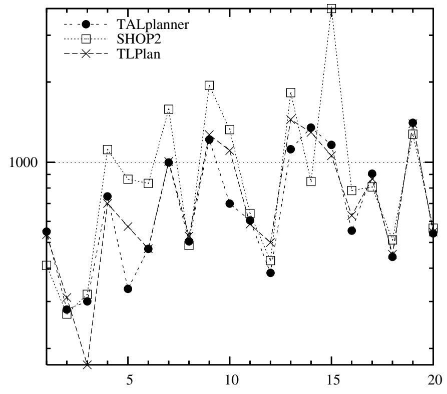
(c) Time (seconds, SimpleTime)



(d) Makespan (SimpleTime)



(e) Time (seconds, Timed)



(f) Problem-Specific Cost Metric (Timed)

Figure 9.8: Results for IPC-2002 ZenoTravel Problems

it appears that only TALplanner and SHOP2 generate concurrent plans for the STRIPS variation of ZenoTravel. In terms of makespan (not shown here), TALplanner generates significantly better plans than SHOP2, which in turns generates plans significantly better than the sequential plans created by TLPlan and FF.

For the SimpleTime version, TALplanner is more or less consistently better than TLPlan, which now does generate concurrent plans, as well as SHOP2. Again, the results are considerably closer if quality is measured in terms of the number of plan steps (not shown here). Consequently, the TALplanner advantage over TLPlan in terms of plan quality is mainly related to scheduling and better use of concurrency.

For the Timed version, each problem instance was associated with a specific metric: Planners should attempt to minimize $x \cdot \mathsf{total\text{-}time} + y \cdot \mathsf{total\text{-}fuel\text{-}used}$, where $\mathsf{total\text{-}time}$ is the makespan of the plan, $\mathsf{total\text{-}fuel\text{-}used}$ is the amount of fuel used by the plan, and the weights $x \in \{1, 2, 3, 4, 5\}$ and $y \in \{0.001, 0.002, 0.003, 0.004, 0.005\}$ varied between problem instances. TALplanner does not support optimization, and there was no attempt to adapt plan generation to each problem metric. Plan quality was similar across the set of planners, though SHOP2 often generated somewhat worse plans and required significantly more time than TALplanner, which in turn required slightly more time than TLPlan.

A fourth version of ZenoTravel, the Numeric version, was available in the contest but due to lack of time we decided not to compete in this domain.

9.2.3 The Depots Domain

The Depots domain (illustrated in Figure 9.9) contains locations, trucks, hoists, crates that can be moved, and pallets whose locations are fixed. Trucks move crates between any two locations and can carry any number of crates at the same time. Hoists are distributed among the locations and load crates into trucks or stack crates on surfaces (pallets or other crates). The goal is always to bring the crates into a certain configuration of stacks, where each stack is placed on a specific pallet.

STRIPS. The Depots domain is a combination of two other well-known planning domains, the logistics domain and the blocks world. Therefore it seems natural to start by taking a look at existing control rules for those two domains, and to see whether those rules can be combined easily or whether more complex rules are required due to interactions between moving and stacking blocks.

We begin with the blocks world part of the problem. The unbounded blocks world was used as a benchmark domain in IPC-2000, and there TALplanner used a modified version of the rules in Bacchus and Kabanza (2000) which ensure that the planner only adds blocks to "good towers", stacks that are already in their final position and will not have to be dismantled later in order to remove a block at a lower level. Can these rules be reused in the Depots domain? One prerequisite is the availability of temporary storage for all crates, since in the worst case every single stack of crates must be torn down completely before it is possible to start

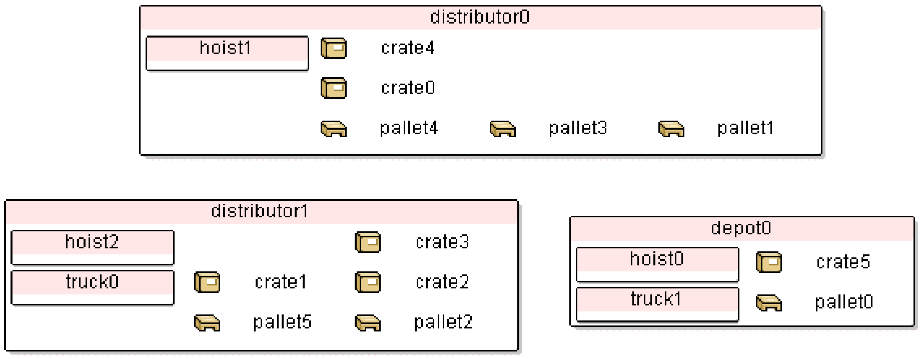


Figure 9.9: A Depots problem instance (STRIPS problem 7)

stacking crates on top of each other. Surprisingly, although there are only a limited number of pallets, trucks can (somewhat counter-intuitively) contain any number of crates, and the planner can use them as storage. Had this been a real-world domain, we would have asked the domain experts whether this could truly be the intention behind the domain, but since it was a competition domain, we could not expect the organizers to change the domain in the middle of the competition. Neither could we ignore this opportunity, since other competitors would surely make use of it. Consequently, we did use trucks as infinite storage facilities, and only minor changes to the logistics control rules were required in order to handle the two separate types of surfaces: Pallets and crates.

Continuing with the logistics part, one simple rule can be reused from the standard logistics domain: “Only unload a crate at its goal location”. Its dual rule, “only load a crate if it needs to be moved”, is not required. The blocks world rules ensure that a hoist does not lift a block unless it needs to be moved, and therefore it is already impossible to load such blocks into a truck.

It remains to ensure that vehicles only drive to those locations where they can be of use. In the standard logistics domain, a truck can drive to another location if there is a package that needs to be picked up or delivered there, but due to the use of stacks of crates in the depots domain, the rule must be modified: A vehicle may drive to a location if (1) there is a crate there that must be moved to another location, (2) there is a crate there that must be stacked differently, or (3) there is a crate in the truck that needs to be at the location, its destination is ready, and there is no other crate that should also be at the same location that the truck has not yet picked up.

SimpleTime. In the SimpleTime version, lifting and dropping crates still takes one unit of time, loading takes three units, unloading four, and driving ten. A few changes were made to ensure mutual exclusion. For example, hoists can only lift one crate at a time. Also, a driving-to fluent was introduced to keep track of where trucks are headed, similar to flying-to in ZenoTravel.

Timed. In the Timed domain, the time required for loading and unloading a crate depends on how powerful the hoist is and on the weight of the crate. The time required for driving between two locations depends on the speed of the truck and the distance between the locations. Again, only minor changes were required to handle the domains, although higher quality plans could certainly have been produced by taking timing into account when determining which hoists and trucks to use.

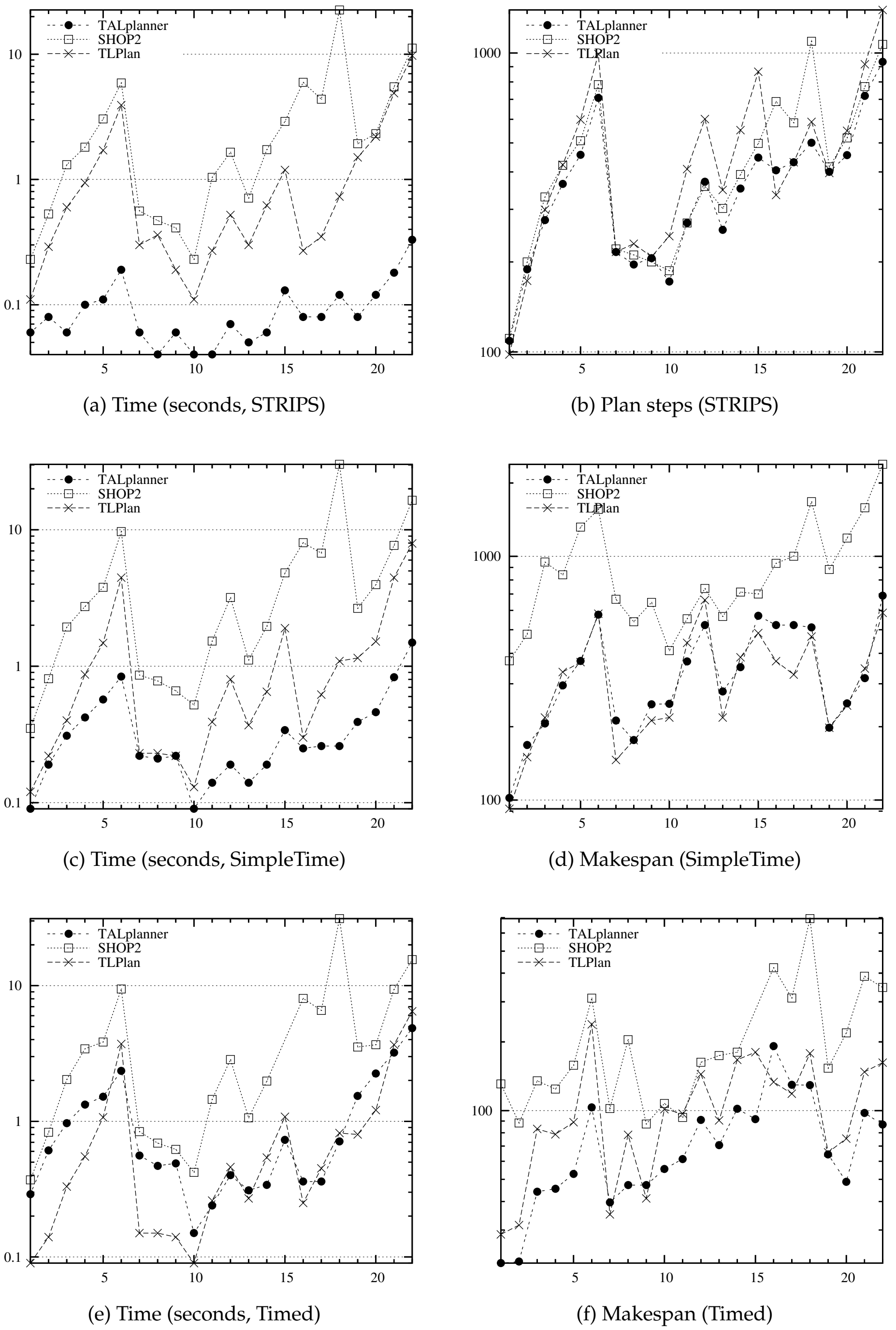
Results. Figure 9.10a shows the time required to solve the STRIPS versions of the 22 larger depots problem instances intended for the three planners in the hand-tailored track of the competition. As can be seen in this figure, TALplanner outperforms SHOP2 and TLPlan by approximately a factor of ten for many problem instances. TALplanner retains much of this performance advantage for the SimpleTime problem instances, as seen in Figure 9.10c, though in the timed problem instances (Figure 9.10e), TALplanner falls back to being slower than TLPlan for the smaller problem instances and approximately as fast for the larger instances.

For the STRIPS version, TALplanner often generates shorter plans than TLPlan and SHOP2, though for some problem instances it generates slightly longer plans. Neither TLPlan nor SHOP2 appear to generate concurrent plans for this domain, giving TALplanner a considerable advantage in terms of makespan (not shown here). In the SimpleTime version, TLPlan has a small makespan advantage over TALplanner, while both planners create considerably better plans than SHOP2. Unlike the ZenoTravel domain, this domain and the remaining domains in the competition did not use problem-specific metrics for the Timed version of the domain, falling back on plain makespan to compare plan quality. Here, TALplanner is again generally better than both TLPlan and SHOP2, though TLPlan does generate better plans for a few problem instances.

9.2.4 The DriverLog Domain

DriverLog (illustrated in Figure 9.11) is yet another logistics domain, this time introducing the concept of truck drivers and road maps. A number of packages are transported between locations by trucks. There are two sets of routes connecting the locations: Links, where trucks travel, and paths, which drivers can walk along when not driving a truck. A truck can only have one driver at a time but can load as many packages as is needed.

Finding Shortest Paths. In the DriverLog domain, vehicles and people must travel along road networks, where different roads may have different costs (lengths) and where taking the shortest path between any two points is essential. Although a shortest path algorithm can be defined using the TALplanner input language, the resulting formulas can be somewhat complicated. Two new functions related to shortest paths were therefore implemented in TALplanner: One for finding the cost



(a) Time (seconds, STRIPS)

(b) Plan steps (STRIPS)

(c) Time (seconds, SimpleTime)

(d) Makespan (SimpleTime)

(e) Time (seconds, Timed)

(f) Makespan (Timed)

Figure 9.10: Results for IPC-2002 Depots Problems

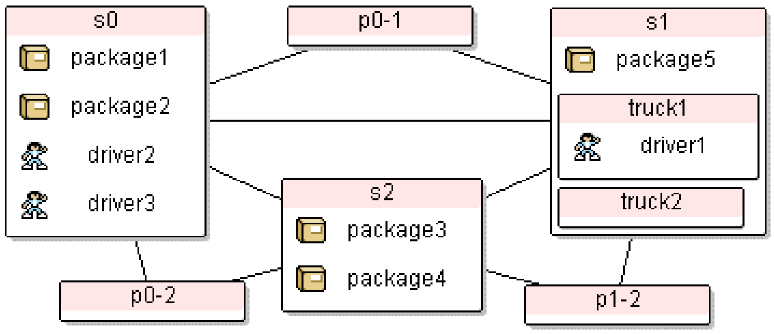


Figure 9.11: A DriverLog problem instance (STRIPS problem 5)

of the shortest path between two given locations, and one for finding the distance to the closest location satisfying a given formula (for example the closest location which is a reasonable destination for a certain truck in the DriverLog domain). These functions also provide a significant improvement in performance, and are useful in many domains where distances are involved, including the Rovers domain seen later (Section 9.2.5 on page 268).

STRIPS. Several control rules used in previous logistics domains were useful for DriverLog with minor modifications. For example, packages should only be loaded into trucks if they need to be moved, and should not be unloaded until they have reached their final destination.

On the other hand, a number of changes were necessary due to the use of road maps. Most importantly, vehicles were previously only allowed to drive to locations that were immediately useful because there were packages to be picked up or delivered. In the DriverLog domain there may only be direct roads between *some* locations (specified by a predicate link(*from*, *to*)), and a truck may have to move through several intermediate locations in order to reach its destination. Consequently the control rules must be relaxed to allow trucks to visit locations that are not useful in themselves. Nevertheless, some degree of goal-directedness is still required. One possible method is to identify for each vehicle the set of locations where the vehicle might be useful, and to require that it chooses one such location and then takes the shortest path to its chosen destination. This method was used in the competition with the help of the built-in shortest path algorithm discussed above and a control rule stating that each step (each invocation of drive or walk) must decrease the distance to the current destination. The following definitions will be explained below:

```
define [$t$] reasonable-truck-location($truck$, $location$):
  // The truck has objects to deliver there.
  $\exists obj$ [ [$t$] in($obj$, $truck$) $\wedge$ goal(at($obj$, $location$)) ] $\vee$

   // All objects have been delivered, and
   // either there's a goal that the truck should be there
   // or there's a goal that the driver should be there
   //    and no goal preventing him from using the truck to drive there.
  (([$t$] all-objects-at-their-destinations) $\wedge$
   (goal(at($truck$, $location$)) $\vee$
   ($\neg$goal($\neg$at($truck$, $location$)) $\wedge$
    $\exists driver$ [ [$t$] driving($driver$, $truck$) $\wedge$ goal(at($driver$, $location$)) ]))) $\vee$

  // There are objects to pick up (modeled as a resource)
  // and either we are already there
  // or no other trucks are already there or on their way.
  (([$t$] available(objects-to-move-at($location$)) $\neq$ 0) $\wedge$
   (([$t$] at($truck$, $location$)) $\vee$
    $\neg\exists truck_2$ [ $truck_2 \neq truck$ $\wedge$ [$t$] $\neg$empty($truck_2$) $\wedge$ [$t$] at($truck_2$, $location$) ] $\wedge$
    $\neg\exists truck_2$ [ $truck_2 \neq truck$ $\wedge$ ([$t$] $\neg$empty($truck_2$)) $\wedge$ [$t$+1] at($truck_2$, $location$) ]))

distfeature driving-distance-between($from$, $to$) :domain integer :link link

mindistfeature driving-distance-to-location-satisfying-formula
  :distfeature driving-distance-between :domain integer

define [$t$] driving-distance-to-reasonable-destination($truck$, $location$):
  driving-distance-to-location-satisfying-formula
    ($location$, $to$, [$t$] reasonable-truck-location($truck$, $to$))
```

A boolean fluent reasonable-truck-location(*truck*, *loc*) is defined in terms of a logic formula, which specifies whether the given location is a reasonable destination for a given truck at the timepoint when it is evaluated. The driving-distance-between function accesses the shortest path algorithm to find the length of the shortest path between *from* and *to*, given that the road links are specified by the link predicate. The driving-distance-to-location-satisfying-formula function accesses another version of the shortest path algorithm and is used in driving-distance-to-reasonable-destination in order to find the shortest distance from *location* to any location *to* that satisfies reasonable-truck-location. Since all links have the same cost, it is then sufficient to require that whenever a truck moves, its driving-distance-to-reasonable-destination decreases.

Further changes were required due to the use of drivers. There may not be drivers for all trucks, so packages should not be loaded into a truck until the planner knows the truck will have a driver. Drivers should not disembark if there are still packages in the truck, or if there is a goal that the truck must be somewhere else. Drivers may have to walk along paths in order to reach a truck, so just like trucks, drivers must select one useful destination and then take the shortest path to their chosen destinations.

Additional control rules ensure that multiple trucks do not choose the same destination unnecessarily, and that multiple drivers do not choose to walk to the same location.

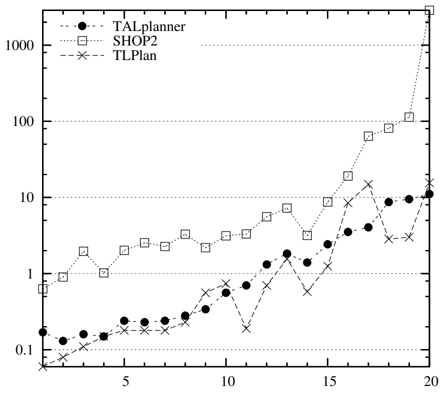
SimpleTime. In the SimpleTime version, loading and unloading objects takes two units of time, driving takes ten units, and walking takes twenty units. The operators are changed accordingly, and a going-to fluent is introduced to keep track of drivers and trucks that are moving towards a new location but have not yet arrived. A few minor adjustments must be made to the control rules.

Timed. In the Timed version, the time required to walk or drive between two locations is determined by a pair of functions specified in each problem instance. Since individual road segments can have different lengths, the method we used to ensure drivers and trucks used the shortest path to their current destination is no longer sufficient, and must be modified slightly. Other than this, there are no major changes for the Timed version.

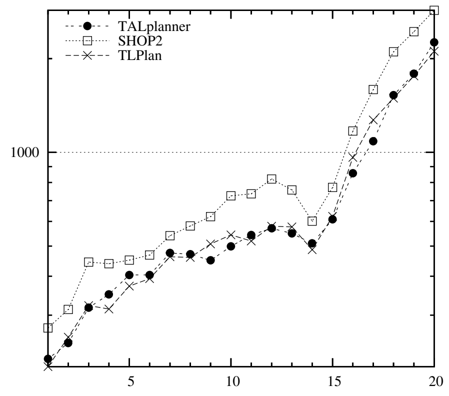
Results. TALplanner generates plans quite quickly for STRIPS instances but has somewhat worse performance for SimpleTime and Timed problems, with TLPlan generally taking the lead (Figure 9.12). For the STRIPS version, TALplanner and TLPlan generate plans of approximately equal quality in terms of plan steps, though once again TLPlan does not generate concurrent plan and is therefore at a disadvantage in terms of makespan (not shown here). SHOP generates somewhat longer plans than TLPlan, but does generate concurrent plans. For the SimpleTime version, no planner appears to consistently produce better plans than any other. In the Timed version, the plans generated by TALplanner generally have a longer makespan than those created by TLPlan, though they do tend to contain fewer actions (not shown here).

9.2.5 The Rovers Domain

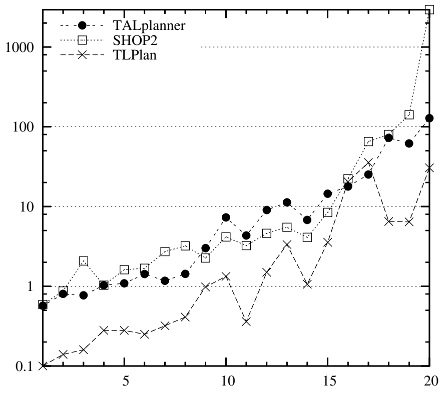
The Rovers domain simulates a simplified planetary exploration expedition. A lander vessel carries a number of rovers to the planet surface and provides a communication link back to Earth. Each rover has a subset of the general capabilities: retrieving soil samples, retrieving rock samples and capturing images using cameras that support different imaging modes. The cameras are mounted on the rovers, as are storage compartments, one for each rover, which can hold one soil sample or one rock sample. Data from a sample must be sent to the lander by a communication link. All missions revolve around navigating waypoints on the surface of the planet to collect samples and take images of specified objectives that are only visible from certain waypoints. The terrain may prevent rovers from going directly between two waypoints and different rovers handle different terrain so a list of routes each rover can use is provided.



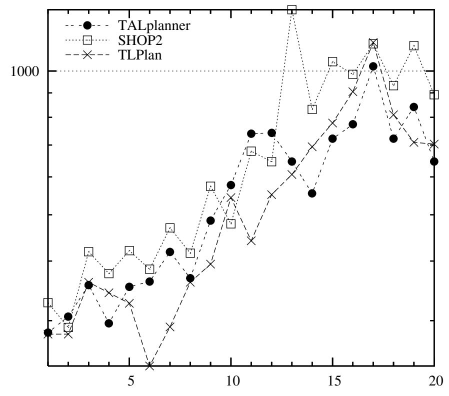
(a) Time (seconds, STRIPS)



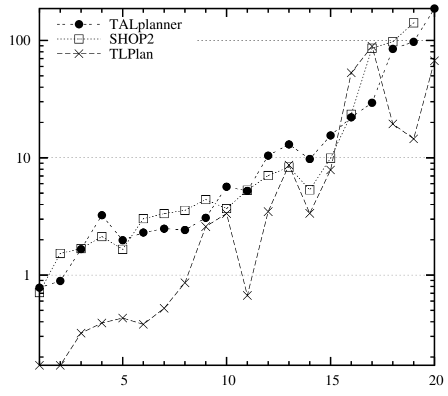
(b) Plan steps (STRIPS)



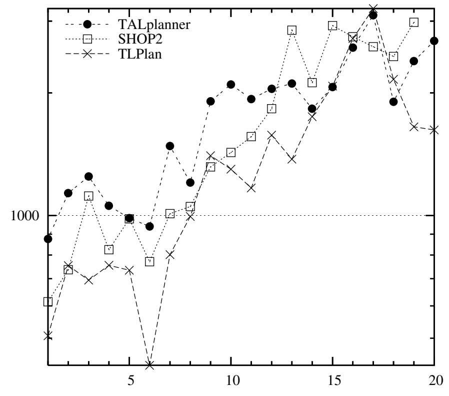
(c) Time (seconds, SimpleTime)



(d) Makespan (SimpleTime)



(e) Time (seconds, Timed)



(f) Makespan (Timed)

Figure 9.12: Results for IPC-2002 DriverLog Problems

STRIPS. Following a control scheme similar to the one used in DriverLog, we limit the movements of rovers to locations where they can perform some useful action like collecting a rock sample or capturing an image. The problem of finding a path from one waypoint to another is also solved in the same way as in DriverLog, except that each rover has its own set of routes between waypoints.

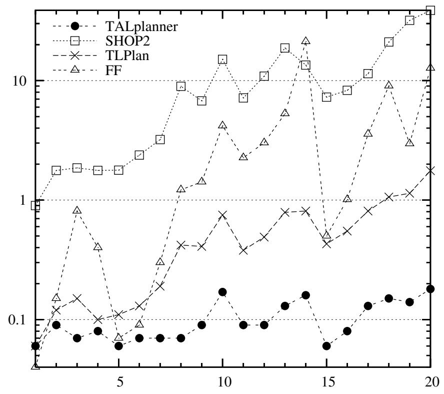
SimpleTime. The changes in the SimpleTime version are trivial: Operator durations are changed, a few mutual exclusion relations need to be enforced, and a new fluent calibrating(camera) keeps track of whether a certain camera is being calibrated.

Timed. The Timed version introduces the concept of energy, where each rover has a limited amount of energy and each action it does consumes some of the energy. This is similar to the use of fuel in the ZenoTravel domain, but there is also a major difference: The rovers have been equipped with solar panels that recharge the rover, but only some of the waypoints that a rover can go to are directly exposed to the sun, which is a requirement for the solar panels to work. The airplanes in the ZenoTravel domain can refuel anywhere, and so fuel usage is only relevant in terms of minimization of resource usage, whereas a rover that uses its energy unwisely can get stuck in the shade, unable to do anything or go anywhere. To prevent this we can either let the planner backtrack and search for a better plan, or we can introduce stricter rules that keep energy levels in mind when deciding what a rover is allowed to do. The latter approach is taken below.

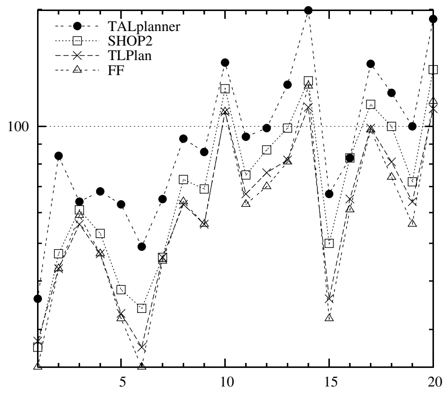
The critical point is when a rover does not have enough energy to reach a waypoint in the sun and recharge. Using the shortest path algorithm it is possible for a control rule to determine the distance to the closest waypoint that is exposed to the sun. In addition to all waypoints that were previously allowed, it is also reasonable for a rover to go to a waypoint that is exposed to the sun if the rover does not have enough energy to perform an action and then go recharge, or if there do not exist any other waypoints that are both affordable and reasonable to visit.

Results. Timing results for the Rovers domain varied significantly between different planners (Figure 9.13). In the STRIPS domain, TALplanner took the lead over TLPlan by a factor of 10 for the larger problem instances, while SHOP2 was slower by another factor of 20. FF generally beat SHOP2 despite being a fully automated planner, but rarely came close to the performance of TLPlan and TALplanner. For the SimpleTime version, the margin between TALplanner and TLPlan was narrowed to a factor of 3 for the larger problem instances, and for Timed instances, TALplanner was only slightly faster than TLPlan while SHOP2 remained considerably slower.

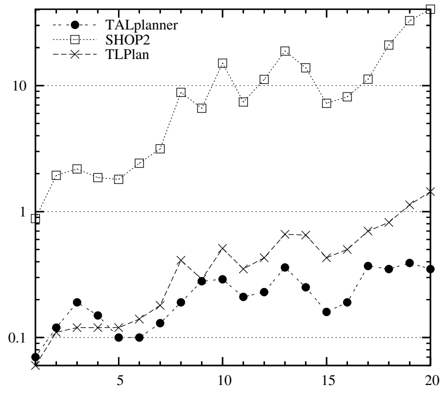
Plan quality results for this domain are quite interesting. For the STRIPS version, TALplanner generated plans of consistently somewhat worse quality compared to its competitors, both in terms of plan steps and in terms of makespan. For SimpleTime instances, plans were of similar length and almost identical execution time. For Timed instances, though, TALplanner generated plans with almost



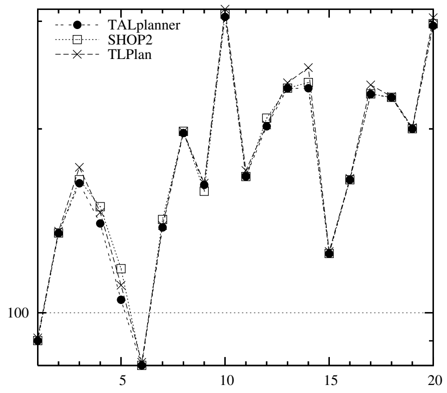
(a) Time (seconds, STRIPS)



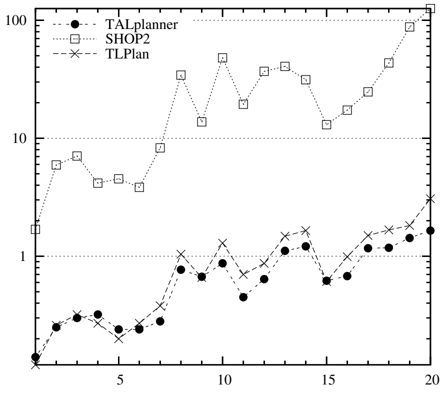
(b) Plan steps (STRIPS)



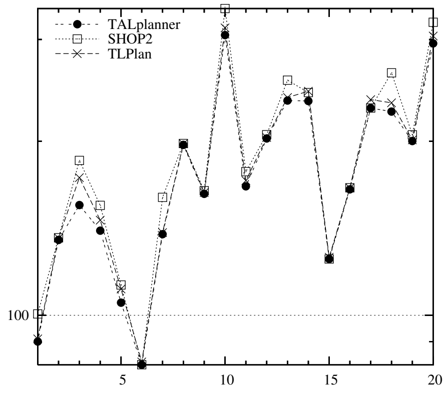
(c) Time (seconds, SimpleTime)



(d) Makespan (SimpleTime)



(e) Time (seconds, Timed)



(f) Makespan (Timed)

Figure 9.13: Results for IPC-2002 Rovers Problems

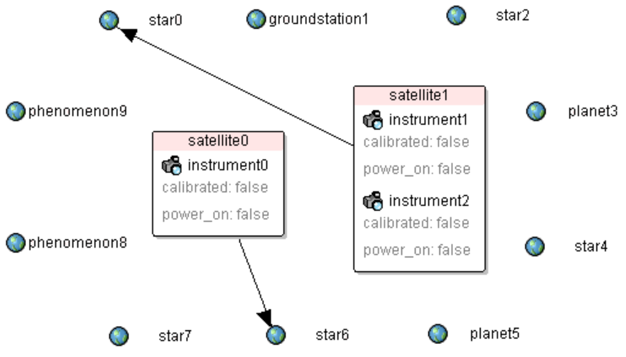


Figure 9.14: A Satellite problem instance (STRIPS problem 4)

twice as many actions (not shown here) but still generally managed to stay within a shorter makespan than the other planners. Further analysis reveals this to be due to a simple systematic mistake in the Rovers domain which should have been easily discovered: When there were fewer tasks left to be performed than there were rovers, the superfluous rovers would move around aimlessly, occasionally recharging to be able to keep moving. This does not affect the execution span of the plan, since the remaining rovers were still able to perform their tasks without interference.

9.2.6 The Satellite Domain

In the Satellite domain a number of satellites orbit the Earth, each equipped with a set of scientific imaging instruments. The satellites turn in space, targeting stars, planets and interesting phenomena to capture images of them using different instrument operation modes. These modes can include regular or infrared imaging and spectrographic or thermographic readings but are different for each problem. The planner's task is to schedule a series of observations so that the satellites are used efficiently. Figure 9.14 shows a small example problem instance, with arrows showing the directions in which the satellites are pointing.

Directions are not represented as explicit coordinates. Instead, satellites can turn to a new direction by giving the turn-to operator an argument specifying the star, planet or phenomenon that the satellite should point to. Instruments first need to be activated using switch-on, then calibrated at a calibration target with the calibrate operator before they can capture images using take-image. Each satellite has only enough power to operate one instrument at a time, so switching active instruments is always initiated by the switch-off operator to deactivate the first instrument.

Satellite: STRIPS

Since the task consists of collecting a number of images, we begin by restricting the use of take-image to images that are mentioned in the goal.

```
control :name "only-take-pictures-of-goals"
[$t$] ¬have-image($direction$, $mode$) ∧ [$t$+1] have-image($direction$, $mode$) →
goal(have-image($direction$, $mode$))
```

The next step is to restrict the directions in which satellites turn to those that may actually help in collecting the images. The task is split into a control rule, only-point-in-goal-directions, and a definition of goal directions. A satellite is allowed to turn towards a direction to take a picture, to calibrate an instrument or if a goal specifies that the satellite should point in the direction and there is no more work left to do.

```
define [$t$] goal-direction($satellite$, $direction$):
  [$t$] take-image-possible($satellite$, $direction$) ∨
  ∃$instrument$ [
    [$t$] power-on($instrument$) ∧ ¬calibrated($instrument$) ∧
    [$t$] calibration-target($instrument$, $direction$) ∧ on-board($instrument$, $satellite$) ] ∨
  goal(pointing($satellite$, $direction$)) ∧ [$t$] all-images-collected
```

The take-image-possible function checks not only if an image is to be collected but also that it has not already been taken and that the satellite has the necessary instrumentation ready. If the active instrument is not calibrated, the satellite may first have to turn towards another direction and calibrate it.

```
define [$t$] take-image-possible($satellite$, $direction$):
  ∃$mode$ [ goal(have-image($direction$, $mode$)) ∧
            [$t$] ¬have-image($direction$, $mode$) ∧
            ∃$instrument$ [
              [$t$] power-on($instrument$) ∧ calibrated($instrument$) ∧
              [$t$] on-board($instrument$, $satellite$) ∧ supports($instrument$, $mode$) ]]
```

The switch-on and switch-off operators are still not regulated by control rules and the planner quickly takes up the habit of repeatedly flipping the power to different instruments on and off. Once an instrument has been powered on and calibrated, using it as much as possible before switching to another instrument seems reasonable. A usefulness function, putting a value on the usefulness of a particular instrument, helps decide which instrument to power on first.

```
define [$t$] usefulness($instrument$):
  $value$ ($t$, $sum(<$mode$>, [$t$] supports($instrument$, $mode$) ∧ mode-needed-for-goal($mode$), 1))

define [$t$] mode-needed-for-goal($mode$):
  ∃$direction$ [ goal(have-image($direction$, $mode$)) ∧ [$t$] ¬have-image($direction$, $mode$) ]
```

Add one to the usefulness score of an instrument for each imaging mode that it supports and that is needed in some goal. This score is then used in a control rule that chooses a satellite's most useful instrument, if it has any.

```
control :name "use-the-most-useful-instrument"
  [t] ¬power-on(instrument) ∧ [t+1] power-on(instrument) →
  [t] usefulness(instrument) > 0 ∧
  ¬∃satellite, instrument_2 [
    [t] usefulness(instrument_2) > usefulness(instrument) ∧
    [t] on-board(instrument, satellite) ∧ on-board(instrument_2, satellite) ]
```

Switching off an instrument is only allowed if the instrument is no longer required.

```
control :name "do-not-switch-instrument-off-if-you-do-not-have-to"
  [t] power-on(instrument) ∧ [t+1] ¬power-on(instrument) →
  [t] ¬∃mode [ supports(instrument, mode) ∧ mode-needed-for-goal(mode) ]
```

We have run out of more or less obvious improvements, but analyzing the planner output reveals one remaining inefficiency: The satellites often simultaneously decide to turn to the same direction because a picture needs to be taken in that direction, despite the fact that only one satellite needs to take the picture. This is similar to the situation in the ZenoTravel domain where a number of aircraft may concurrently choose to pick up the same passenger, but there are some differences due to the fact that the *only* reason for a satellite to point in a certain direction is in order to calibrate itself or take an image, which makes the task somewhat easier.

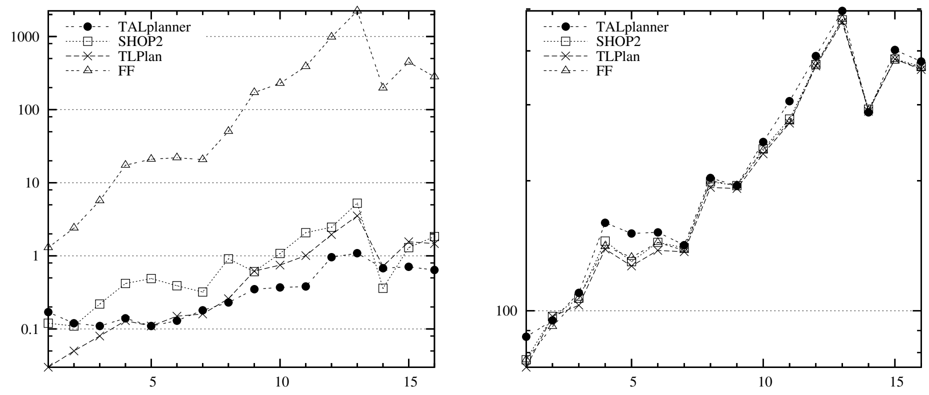
Therefore this problem can be solved in a different way, using a resource for mutual exclusion. This resource, called point-towards(*direction*) and having a capacity of 1, can be borrowed temporarily by turn-to for the duration of the turn. If one satellite turns towards a specific direction d, no other satellite can turn towards d without causing a resource conflict.

This still leaves one problem: When the first satellite has finished turning, it no longer owns the point-towards(d) resource and therefore another satellite can immediately start turning towards d. It is no longer possible for more than one satellite to turn towards the same direction at once, but while the first satellite is taking pictures, other satellites can turn to that direction one by one, until finally all the desired pictures have been taken in that direction and goal-direction sees that there is no longer any valid reason to point towards d. This can be solved either by changing the definition of goal-direction or by letting take-image borrow the same resource.

Clearly, this type of "swarming" problem occurs quite often in concurrent domains and a more principled solution should be investigated in the future.

Satellite: SimpleTime

The SimpleTime version changes the duration of some operators. Turning takes five time units, switching an instrument on takes two units, calibrating it takes five



(a) Time (seconds, STRIPS) (b) Plan steps (STRIPS)

Figure 9.15: Results for IPC-2002 Satellite Problems: STRIPS

units and taking a picture takes seven units. A couple of helper fluents, turning-towards, calibrating, have-image-generalized (an image exists or is being taken) and power-on-generalized (power is on or a switch-on action is being executed) keep track of actions that have begun but not completed. The affected control rules are updated accordingly.

Satellite: Timed

The Timed version of the Satellite benchmark domain includes two new functions. The calibration-time specifies the time required to calibrate, while the slew-time function represents the time required for a satellite to turn between two directions. Neither of these changes prompts any significant changes to the SimpleTime control.

Satellite: Results and Discussion

The Satellite domain does not provide a real challenge as long as the planner is only trying to find a correct solution to each problem instance. Finding a short solution is harder, especially in the Timed version, and would require additional analysis to determine in which order images should be collected and which satellites should be used for each image. Doing this using control rules seemed a bit like overkill, especially since we had not yet created control rules for the complex UMTranslog-2 domain. For this reason, we decided to be satisfied with what we had done so far.

For the STRIPS version, TALplanner turned out to generate plans approximately as quickly as the other planners (Figure 9.15). The plans were slightly longer than those of TLPlan, SHOP2 and FF in terms of plan steps, but generally shorter in terms of makespan (not shown here).

In the SimpleTime domain, TLPlan was considerably faster, but for the Timed version, TALplanner was once again approximately as fast as TLPlan and faster than SHOP2 (Figure 9.16). However, we were quite surprised to find out that the plans we generated were considerably worse than those from the other planners in terms of makespan – worse by up to a factor of 10!

After the contest, we were informed of the reason, or at least the main reason: The automatic problem generator that created the problem instances randomized the slew times between every pair of directions and did not check for geometrical consistency that would be present in a real world situation. We had subconsciously assumed that the problem instances satisfied the triangle inequality, but this was not the case, and the other planning teams had discovered this. For example, in handcoded problem 14, turning a satellite directly between **phenomenon86** and **groundstation4** takes 82.860 units of time, while turning it through two carefully selected intermediate directions requires 1.183 units of time. But going through intermediate directions also requires a greater number of actions, which can clearly be seen in the Figure 9.16 where TALplanner consistently uses considerably fewer actions than its competitors for the Timed domain.

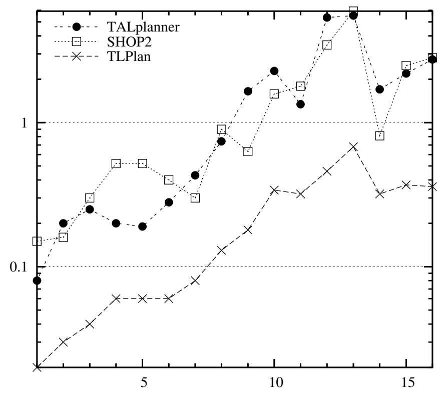
Taking this into consideration and once again using the built-in shortest path algorithm yields significantly shorter plans when plan length is measured by the time point at which the goals have been satisfied. Another potential improvement would be to change the last clause in goal-direction to allow satellites to turn towards a direction specified in the goals as soon as one has started taking the last picture, rather than waiting until one has finished taking the last picture.

9.2.7 The UMTranslog-2 Domain

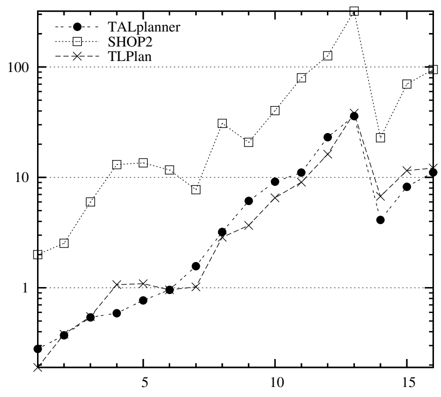
The UMTranslog-2 domain is another logistics domain, but with 14 distinct object types, 38 predicates, 24 functions and 38 operators, its size and complexity is incomparable to the previously encountered logistics domains in the contest.

Since the formal domain definition was the only information provided about the domain and there was no high-level description, we had to work out all the information about the domain from the PDDL definition. This was not a major problem for the previous domains, since they were generally quite simple and easy to understand, but it did give us some problems in UMTranslog-2. A significant amount of time was spent trying to determine exactly how packages were allowed to move and how they can be loaded into and unloaded from various kinds of vehicles. In retrospect, it would probably have been better to do as some other teams did: Skip the UMTranslog-2 domain completely and spend that time on the Numeric and Complex versions of the other domains.

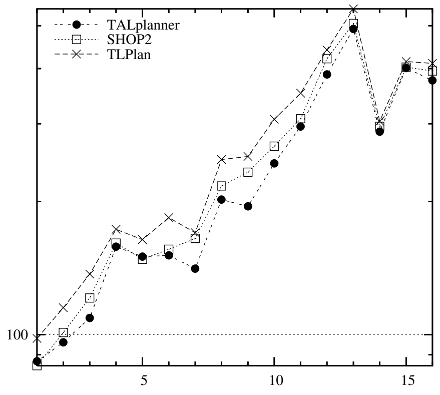
The domain. Trucks, trains or aircraft transport packages between locations but they must follow strict movement patterns. A few locations are transportation hubs, some are transportation centers while the rest are ordinary locations. A package is only allowed to move up and down through this hierarchy once and only



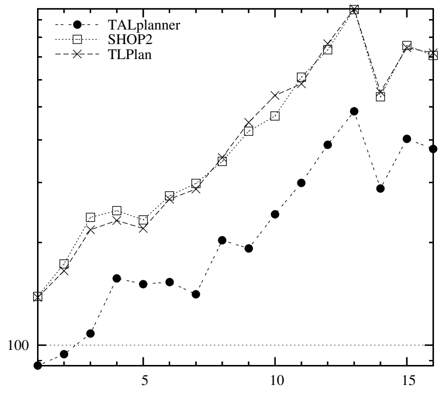
(a) Time (seconds, SimpleTime)



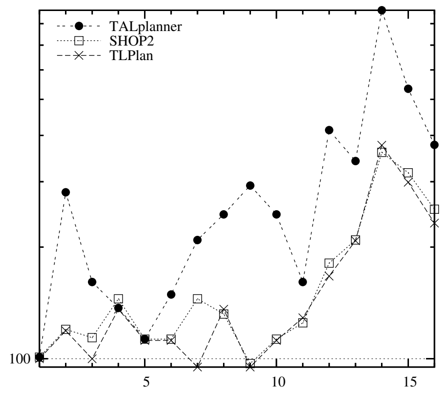
(b) Time (seconds, Timed)



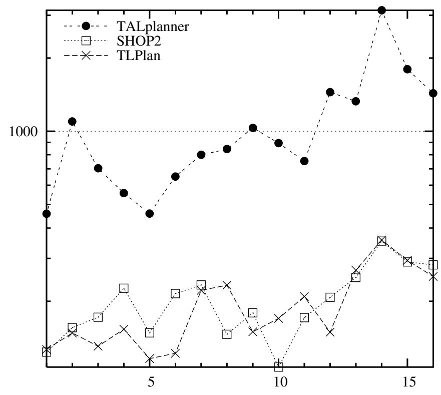
(c) Plan steps (SimpleTime)



(d) Plan steps (Timed)



(e) Makespan (SimpleTime)



(f) Makespan (Timed)

Figure 9.16: IPC-2002 Satellite Problems: SimpleTime and Timed

move between two locations in the same layer once. The longest possible route for a package is thus from an ordinary location to a transportation center to a hub to another hub to a transportation center and finally to another ordinary location.

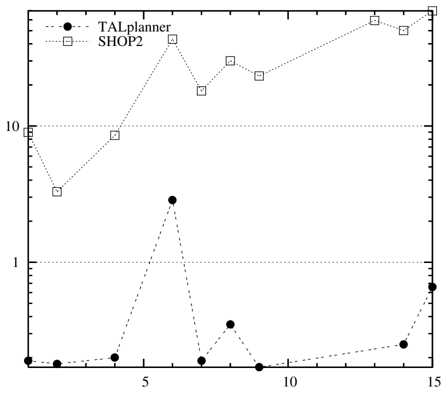
The domain groups locations into cities, which are then grouped in regions. Trucks travel between any two locations in the same city or by an existing road route between two cities. Trains and planes always use predefined routes between transportation centers and hubs. A great number of restrictions further complicate movements. Packages must be compatible with the vehicle they are loaded into, the vehicle must have enough free space, not be loaded too heavily and not be wider, longer or higher than the route and destination location accepts. Finally, the locations, vehicles and routes must all be available for use.

Control rules. As in previous domains, we specify what a reasonable location is and limit vehicle movements to destinations that are reasonable. A truck might want to pick up or deliver a package at the location or, if the truck cannot reach the goal location of the package, unload the package at a transportation center to be picked up by another vehicle. Our control rules do not allow trucks to pick up several packages. This makes finding optimal solutions impossible in the general case but simplifies the search for acceptable solutions a great deal. There is an imminent risk that any other packages the truck is carrying will end up at the wrong location if it is allowed to travel about, picking up more packages along the way. Since all packages must move according to the specified pattern of transportation centers and hubs, moving a package that has once arrived at a location that is not a transportation center is not allowed and the package will be stuck there. Restricting trucks to picking up one package at a time avoids this problem.

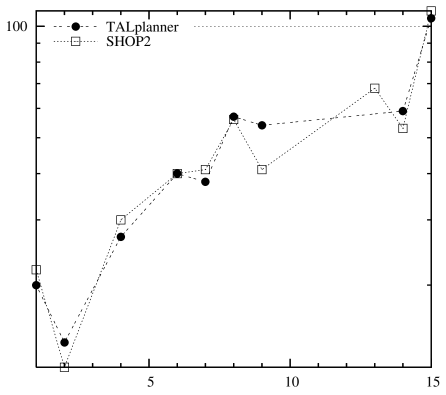
There is also a large group of loading and unloading rules controlling, among other things, the opening or closing of valves and doors and loading or unloading of packages. This part of the problem would perhaps be more succinctly solved by a HTN-style planner or by some form of macro operator, since it involves specific sequences of actions that must always be performed in the same order. For example, loading a package into an aircraft always involves attaching a conveyor ramp, opening the airplane door, loading the package, closing the airplane door, and detaching the conveyor ramp, in exactly that order. Nevertheless, writing control rules ensuring the proper order is not difficult – merely somewhat tedious.

Finally, packages are only loaded into vehicles that are actually able to take them to a useful location.

Of the 15 contest problems provided, all created automatically by a problem generator, only ten were actually solvable. The remaining five were unsolvable for different reasons. This may have been intentional, and the ability of a planner to terminate in reasonable time given an unsolvable problem is certainly a valuable quality, as many real world problem instances might be unsolvable. Though determining solvability may be impossible in the general case, it is certainly possible for sequential classical planning due to the use of a finite state space and a finite



(a) Time to solve solvable problem instances



(b) Plan steps for solvable problem instances

Figure 9.17: Results for IPC-2002 UMTranslog-2 Problems

number of possible action instances together with the fact that the applicability of an action only depends on the invocation state as opposed to the entire history of preceding states.

Creating control rules and meeting the contest deadline left no time to get the domain working with concurrent planning. Instead, we had to make do with sequential planning.

Given more time, the set of control rules could definitely be improved. If planning speed is less of an issue, more search can be allowed and higher quality plans generated. More and better problem instances would be needed as guidelines when developing better control rules since the contest problems did not make full use of the intended transportation scheme with transportation centers and hubs.

Nevertheless, the plans generated by TALplanner were of comparable quality to those generated by SHOP2, the only other hand-tailored planner where the team took the time to generate control knowledge for this complex domain (Figure 9.17). TALplanner was also considerably faster than SHOP2 when generating these plans.

Chapter 10

Discussion

In the final chapter of the planning section, it is time to take a step back for a critical view of the work that has been done, the decisions that were made, and the lessons that have been learned along the way. We start by inspecting the initial reason for developing a new planner.

10.1 TALplanner and the WITAS Project

The original decision to take the step from the field of reasoning about action and change (RAC) to the field of planning was grounded in the necessity for a fast and expressive action planner in the WITAS unmanned aerial vehicle (UAV) project (Doherty et al., 2004; Doherty, 2004; Merz, 2004; Doherty et al., 2000).

Developing an autonomous UAV is clearly a long-term project, where a great number of different functionalities must be integrated into a single coherent architecture with support from higher level deliberative systems as well as lower level control systems responsible for ensuring that the UAV remains stable under varying conditions. Though it would perhaps be possible in theory to first design a complete architecture and then commence working on all subsystems in parallel, integrating the entire system when the subsystems are ready, an incremental approach is far more likely to yield a satisfactory result. Such an approach involves beginning with an architecture comprising the most essential subsystems, working on these subsystems and their integration until the system is sufficiently stable, and then incrementally adding new subsystems and improving existing ones as required in order to extend the set of tasks that the autonomous system can perform.

Some of the first systems to be integrated into the WITAS UAV include the basic control systems that allow the UAV to fly autonomously (and later also take off and land autonomously, which is a considerably more difficult task), multiple navigation systems including GPS and inertial navigation allowing the UAV to determine where it is relative to a given coordinate system, and image processing systems that identify ground vehicles and allow the UAV to track such vehicles along road networks. The current UAV architecture has also evolved to include a framework for storing and accessing qualitative knowledge representing various aspects of the environment that the UAV inhabits (Heintz & Doherty, 2005, 2004a, 2004b), a path planner allowing the system to generate (and then fly) a path between its current location and its destination given a map containing buildings and other objects to be avoided (Pettersson, 2003; Pettersson & Doherty, 2004), and a chronicle recognition system that can be used to identify and classify the behaviour of other agents in the environment (Heintz, 2001).

The system now provides a useful set of high-level operators that could be used by an action planner, and consequently the time has come to integrate TALplanner into the UAV architecture. In the first phase, this will merely require some additional work in terms of interfacing the planner with the CORBA-based system architecture, ensuring that the higher interface levels of the system can and do access the planning functionality when this is relevant to the current mission, and of course also creating a TAL-based model of the UAV domain at an appropriate level of abstraction. Thereafter, another research phase will begin, aimed partly at determining which parts of UAV missions are best solved by traditional action planning as opposed to more "hard-coded" solutions and partly at determining which aspects of the planner will need to be developed further in order to provide better functionality to the UAV system. We expect the most challenging issues to arise in the area of uncertainty: The UAV domain involves a great deal of uncertainty regarding observed values as well as regarding the true effects of an action, and exogenous events must be taken into account. Plans may have to support sensing actions and may have to repeat actions until a desired result is achieved. Also, the plan execution framework may have to be taken into account in the planning phase, with explicit support for safety conditions. All of these are interesting future research issues, and will also be discussed under "Future Work" below.

It remains to be seen whether all extensions that will be required in the future can be cleanly integrated into the current TALplanner framework or whether more extensive modifications will be required. Regardless of the eventual outcome, TALplanner has turned into an intensely interesting research topic in its own right.

10.2 Hand-Tailored versus Fully Automated Planning

As discussed previously, there is no binary distinction between hand-tailored and fully automated planning. Many "fully automated" planners can be tuned for particular problems using configuration settings that affect various parameters controlling the planning algorithm, and some may even be dependent upon such settings for reasonable performance, while some "hand-tailored" planners can assist in the generation of the secondary domain-specific information they require as input. Nevertheless, it must be admitted that there is in general an additional effort involved in using a hand-tailored planner.

Some may use this as a reason for questioning the existence of hand-tailored planners, preferring to unconditionally follow a "pure" approach where a planning algorithm only requires an absolute minimum of information in order to solve a planning problem. Though we agree that fully automated planning is certainly an interesting and worthy field of research, we believe there are several reasons why hand-tailored planners may in some cases be a better short-term approach for an applied research project and may in fact always be a step ahead of fully automated planners in certain respects.

Our first argument relates to the fact that it may not be immediately obvious to what extent a certain kind of information would be useful for a given planning algorithm. Perhaps introducing this information would improve performance by orders of magnitude for many common domain types, or perhaps the impact would be negligible. In such cases, a hand-tailored planner can be seen as a temporary step on the way towards full automation, where only that aspect of the planner that makes use of the new information is implemented. If the information did indeed turn out to be of use, the project can then proceed by implementing the information-gathering aspect of the planner. This approach, which was used to test whether introducing state invariants would improve the performance of the TALplanner formula optimizer, saves time and helps reduce the amount of time spent researching what may turn out to be a dead end.

Even if it is in fact apparent that a certain kind of information would be useful, it may be the case that the proper knowledge for a domain is immediately obvious to a human but difficult to extract for a machine – or, more realistically, that finding this information involves a not completely negligible amount of work for a human, and that it requires a comprehensive understanding of the problem domain that we cannot realistically expect to achieve algorithmically in the near or medium term.

If the gain from using this information is sufficiently large for a particular planning application, either in terms of speed or in terms of plan quality, the decision between having or not having the information available may not be difficult. In our experience with the temporal control formulas used by TALplanner, this has often been the case.

Permitting the use of additional information to guide a planner may also alleviate the need to restrict the expressivity of its domain definition language. There are a multitude of interesting domain analysis techniques that assist fully automated planners in reducing the complexity of the planning task or in finding heuristics or other forms of guidance relevant to a particular fully automated search algorithm, but in many cases such techniques appear to be founded on the assumption of single-step operators, sequential planning, conjunctive preconditions, and/or the ability to generate all ground instances of all facts and operators (Nebel et al., 1997; Haslum & Jonsson, 2000; Fox & Long, 1998; Cresswell et al., 2002; Fox & Long, 2000b, 2000a, 2002; Gerevini & Schubert, 1998, 2000; Scholz, 2000; Rintanen, 2000b).

Allowing the domain designer to provide a certain amount of guidance will facilitate the task of stepping outside of the classical framework and preparing the ground for richer and more expressive models of real-world planning domains.

10.3 Using Control Rules

If we accept the use of hand-tailored information as input to a planner, there is still a question of what shape this information should take. TALplanner uses a set of control formulas that must be entailed by the final solution generated by the planner. Our article on TALplanner in the third International Planning Competition (Kvarnström & Magnusson, 2003), most of which is included as part of Chapter 9, includes a number of domain-dependent control rules for the competition domains. Rather than presenting an exhaustive list of pre-packaged control rules, we attempted to place more emphasis on explaining the incremental analysis process that eventually leads to the final formulas, going into particular detail for the ZenoTravel domain.

As could be seen in these examples, control rules are often simple, natural common-sense rules, and not very difficult to generate given some basic knowledge about the planning domain. Some rules are more complex, but still not difficult to understand or verify once someone has spent the effort to generate them. And then, unfortunately, there are a few rules that are quite unintuitive, rules that are too complex to be easily understood, and rules that occasionally forbid optimal plans.

To some extent, such rules might be avoided by gaining more experience in good practices for writing control rules, by extending the expressivity of the language in which control rules are written so that complex conditions can be expressed more succinctly or in a more natural manner, or simply by spending a little bit more time on the control rules than was available during the planning competition. However, another important cause for the complexity of certain rules is most likely that we are attempting to express all search control knowledge in the same way: As control rules that prune the search tree to such a great extent that even a simple depth first search algorithm is sufficient for efficiently finding good plans in the remainder of the tree.

Not all search control knowledge can easily be expressed in this manner, but this certainly does not mean that control rules should be abandoned altogether. Instead, what we learn from this experience is that control rules might not be the one and only multi-purpose planning tool that will efficiently and easily solve all our planning problems. Just like one would expect, they are one very useful tool that deserves a place in our toolbox but should be combined with other approaches to planning.

To mention one rather obvious example, it would be possible to devise a heuristic forward-chaining planner whose search tree would be pre-pruned using control rule techniques from TALplanner. Control rules could be written to exclude plans where the heuristic gives a suboptimal result, potentially providing plans that are closer to optimal, and even for domains where the heuristic search function provides good plans it may often be more efficient to state a number of constraints as explicit control rules. This would also reduce the need to use extremely strong control rules that provide strong guidance but may prevent the generation of optimal plans. For example, one of the standard logistics control rules states that no packages should be unloaded from a plane until the final destination city has been reached. If this rule is removed, packages will be unloaded from airplanes randomly. If the rule is included in its current shape, it unfortunately also prevents packages from being moved between airplanes at any time, even if this would lead to better overall performance. A well-chosen heuristic function would most likely yield better plans in such cases.

Also, we have noticed a tendency to generate unnecessary plan steps in certain concurrent domains, where attempting to exclude these plan steps instead leads to a decrease in flexibility and a potentially longer makespan. The Rovers domain is a case in point. When a rover was not needed for a specific task, our control rules allowed it to drive around aimlessly, which increased the number of actions in each plan but may have shortened makespans slightly: When the planner saw the need for a rover at a certain point, one of the free rovers might already be there. Integrating a technique such as PbR, Planning by Rewriting (Ambite, 1998; Ambite et al., 2000; Ambite & Knoblock, 2001), would allow such plans to be rewritten in a post-processing step where unnecessary actions would be removed. Though this particular case might be viewed as a patch for bad control rules, it would nevertheless be interesting to investigate the applicability of this combination for a larger variety of domains.

Various combined planning techniques have been considered at least since some time before the second planning competition in 2000, and it has long been clear to us that such approaches should eventually be examined and explored. Before we could start working on this, though, the strengths and weaknesses of control rules had to be explored in more depth. Our work has therefore focused mostly on investigating how far it is possible to take TALplanner in its current shape, with explicit control rules being the only means for controlling the search process. This work has proved rather fruitful in itself, and TALplanner did well in IPC-2000 as well as in IPC-2002.

10.4 Using TAL in TALplanner

One primary requirement for being able to trust the solutions generated by a planner is the use of a well-defined domain modeling language with a formal semantics for planning domains and problem instances.

TALplanner makes use of a modified version of the TAL-C logic for this purpose, a choice that has served us well in most respects. In the first version of

TALplanner, TAL-C provided a suitable semantics for control rules, initial states, goals, and sequential single-step operators. When it was time to extend the planner to support operators with extended duration, and concurrent plans, the formal semantics was already available – and perhaps even more importantly, the intuitions and reasoning behind that semantics, which was developed within the field of reasoning about action and change.

We must admit, though, that the use of an unmodified version of the common TAL base logic $\mathcal{L}(\mathrm{FL})$ has occasionally been somewhat problematic. Providing a proper semantics for recursively defined fluents would require an extension from first-order to fixpoint logic, an extension that has been planned for some time but does not belong within the TALplanner project itself. Modeling arbitrary goals within the logic is trivial as long as only the planning algorithm has to refer to those goals and test whether they are satisfied in a given plan candidate, but control rules need the ability to test whether a formula is *entailed* by the goal, which cannot easily be given a semantics within $\mathcal{L}(\mathrm{FL})$ when disjunctive and existential goals are allowed. Resource effects are given a semantics partly outside the logic using semantic attachment. An extended version of the base logic could remedy most or all of these problems.

The use of TAL in TALplanner is closely related to the use of evaluation rather than progression for control formulas. Both of these approaches have advantages and disadvantages in terms of computational and conceptual complexity.

The process of extracting pruning constraints is conceptually somewhat more complex than the use of progression in TLPlan. This complexity is especially apparent when considering the analysis and optimization algorithms discussed in Chapter 8. Also, certain types of control formulas using the U operator currently cannot be integrated into the incremental pruning constraint framework, making progression a better choice for such formulas.

On the other hand, control formulas that truly require the use of the U (until) operator tend to be rare in our experience, and evaluating the optimized incremental pruning constraints produced by TALplanner is often considerably more efficient than progressing a control formula through all new states generated by an operator, which is a strong point in favor of the use of formula evaluation. These optimizations are more difficult to apply to a progression algorithm, and would lead to a hybrid technique that could no longer truly be called progression.

There is also no need to store a progressed control formula in each search node when evaluation-based control is used, which saves considerable amounts of memory.

In recent work, these two approaches have been unified into a single planner using both progressed tense control rules and evaluated TAL control rules. This enables the user to take advantage of the relative strengths of each approach within any planning domain.

10.5 Future Work

Before considering potential future work within the planning area, let us step back and briefly recapitulate the work has already been presented.

The very first version of TALplanner, implemented during the first two months of the project, was based on ideas from TLPlan. A new underlying formal semantics was developed, based on the use of a modified version of TAL-C providing a declarative semantics for all aspects of the planning process. The action model was significantly extended to allow actions with extended temporal duration and effects at arbitrary timepoints during the execution of the action. TAL-based control formulas were introduced, together with a method for testing such formulas incrementally as a plan was being built. We developed a concurrent version of TALplanner and investigated several techniques for modeling non-interference conditions for concurrent actions. Explicit resource constraints were introduced into the modeling language. A general formula and term optimization framework was developed and applied to incremental pruning constraints as well as other formulas. New operator analysis techniques were added to the optimization framework, and the applicability of existing state transition analysis techniques was investigated. Taken together, these techniques often allow the generation of new preconditions from control formulas and improve the performance of the planner by orders of magnitude for some domains. TALplanner has been empirically tested at various points during its development and has participated in two international planning competitions, winning the highest "distinguished planner" award in the hand-tailored track of IPC-2000.

The main problem during the development of the planner has not been finding interesting and fruitful research issues but prioritizing between all the different directions our research could take.

For example, it has long been clear that there can be much to be gained by combining the use of control rules with other planning techniques into a hybrid planner, as discussed in Section 10.3.

TALplanner currently provides no optimality guarantees, neither in terms of domain-independent measures such as plan length or makespan nor in terms of problem-specific measures specified for each problem instance. In some sense, plan quality can be improved by writing sufficiently good control rules that forbid plans of low quality, but this approach is indirect and provides no optimality guarantees. The very simplest approaches to generating optimal plans might involve applying standard optimal search algorithms to the pruned search tree generated by TALplanner. Some preliminary work has already been done in this area, but further research needs to be done to determine whether this simple approach is sufficient or whether more complex optimization procedures need to be developed. There are also ideas for new types of domain knowledge to be introduced into the planner which should improve the performance of optimal planning by providing constraints applied across multiple plans in addition to the current control rules which only constrain the state sequence generated by a single plan.

In order to apply TALplanner to real-world dynamic domains such as the UAV domain, support for incompletely defined initial states and incompletely specified effects of actions will be very important. Though the planner has not yet been extended in this direction, some of the underlying research related to querying incomplete state structures has already been done by other researchers (Doherty, Łukaszewicz, & Szałas, 2003b, 2003c, 2004a, 2004b, 2000, 2003a; Doherty & Szałas, 2004; Doherty, Grabowski, Łukaszewicz, & Szałas, 2003; Doherty, Kachniarz, & Szałas, 2003).

Many of these topics will be pursued in the future.

10.6 Acknowledgments

TALplanner research has been supported in part by the Wallenberg Foundation, the Swedish Research Council for Engineering Sciences (TFR) and the ECSEL / ENSYM graduate studies program. Special thanks to Patrick Doherty, Patrik Haslum, and Martin Magnusson, who have coauthored a number of papers on which this part of the thesis is based.

Some of the figures in this part of the thesis were generated using TPVis, a domain visualization framework that can animate the movements of objects between different locations and containers in order to give a better instinctive feeling for the quality of the plans generated by a given set of control rules. TPVis was developed by Martin Magnusson after IPC-2002.

Bibliography

Abadi, M., & Cardelli, L. (1996). *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag New York, Inc. See http://www.luca.demon.co.uk/TheoryOfObjects.html.

Alur, R., Feder, T., & Henzinger, T. A. (1991). The benefits of relaxing punctuality. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing (PODC-1991)*, pp. 139–152, Montréal, Canada. ACM Press. Available at http://www.cis.upenn.edu/~alur/Podc91.ps.gz.

Alur, R., & Henzinger, T. A. (1992). Back to the future: Towards a theory of timed regular languages. In *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science (FOCS-1992)*, pp. 177–186, Pittsburgh, Pennsylvania, USA. IEEE Computer Society Press, Los Alamitos-Washington-Brussels-Tokyo. Updated version available at http://www-cad.eecs.berkeley.edu/~tah/Publications/back_to_the_future.ps.

Ambite, J. L. (1998). *Planning by Rewriting*. Ph.D. thesis, University of Southern California. Available at http://www.isi.edu/~ambite/thesis.ps.gz.

Ambite, J. L., & Knoblock, C. A. (2001). Planning by rewriting. *Journal of Artificial Intelligence Research*, *15*, 207–261. Available at http://www.jair.org/contents/v15.html.

Ambite, J. L., Knoblock, C. A., & Minton, S. (2000). Learning plan rewriting rules. In Chien, S., Kambhampati, S., & Knoblock, C. A. (Eds.), *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000)*, pp. 3–12, Breckenridge, Colorado, USA. AAAI Press, Menlo Park, California, USA. Available at http://www.isi.edu/~ambite/2000-aips.ps.

Amir, E. (1999). Object-oriented first-order logic. *Electronic Transactions on Artificial Intelligence*, *3*, 63–84. Available at http://www.ep.liu.se/ej/etai/1999/008/.

Amir, E. (2000). (De)Composition of situation calculus theories. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference* *on Innovative Applications of Artificial Intelligence (AAAI-2000 / IAAI-2000)*, pp. 456–463, Austin, Texas, USA. AAAI Press, Menlo Park, California, USA / The MIT Press, Cambridge, Massachusetts, USA. Available at http://www.cs.uiuc.edu/~eyal/papers/oo-sitcalc-aaai00.ps.

Anderson, C. R., Smith, D. E., & Weld, D. S. (1998). Conditional effects in Graphplan. In Reid G. Simmons, Manuela M. Veloso, S. S. (Ed.), *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-1998)*, pp. 44–53, Pittsburgh, Pennsylvania, USA. AAAI Press, Menlo Park, California, USA.

Artale, A., & Franconi, E. (1998). A temporal description logic for reasoning about actions and plans. *Journal of Artificial Intelligence Research*, *9*, 463–506. Available at http://www.jair.org/contents/v9.html.

Bacchus, F. (2001). The AIPS'00 planning competition. *AI Magazine*, *22*(3), 47–56. See also http://www.aaai.org/Library/Magazine/Vol22/22-03/vol22-03.html and the competition web page at http://www.cs.toronto.edu/aips2000/.

Bacchus, F., & Ady, M. (1999). Precondition control. Available at http://www.cs.toronto.edu/~fbacchus/Papers/BApre.pdf.

Bacchus, F., & Kabanza, F. (1996a). Planning for temporally extended goals. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Conference on Innovative Applications of Artificial Intelligence (AAAI-1996 / IAAI-1996)*, pp. 1215–1222, Portland, Oregon, USA. AAAI Press, Menlo Park, California, USA / The MIT Press, Cambridge, Massachusetts, USA. Available at ftp://newlogos.uwaterloo.ca/pub/bacchus/BKAAAI96.ps.gz.

Bacchus, F., & Kabanza, F. (1996b). Using temporal logic to control search in a forward chaining planner. In Ghallab, M., & Milani, A. (Eds.), *New Directions in AI Planning*, pp. 141–153. IOS Press, Amsterdam, The Netherlands. Available at ftp://newlogos.uwaterloo.ca/bacchus/BKEWSP96.ps.gz.

Bacchus, F., & Kabanza, F. (1998). Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, *22*, 5–27. Available at ftp://newlogos.uwaterloo.ca/pub/bacchus/BKAMAI98.ps.gz.

Bacchus, F., & Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, *116*(1–2), 123–191. Available at ftp://newlogos.uwaterloo.ca/pub/bacchus/BKTlplan.ps.

Bäckström, C., & Klein, I. (1991). Planning in polynomial time: The SAS-PUBS class. *Computational Intelligence*, *7*(3), 181–197.

Blum, A. M., & Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, *90*(1–2), 281–300. Available at http://www-2.cs.cmu.edu/~avrim/Papers/graphplan.ps.

Bonet, B., & Geffner, H. (1998). HSP: Heuristic search planner.. Available at http://www.ldc.usb.ve/~hector/.

Booch, G. (1991). *Object-Oriented Design with Applications*. The Benjamin / Cummings Publishing Company, Inc.

Borgida, A., Brachman, R., McGuinness, D., & Resnick, L. (1989). CLASSIC: A structural data model for objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pp. 58–67, Portland, Oregon, USA.

Brachman, R., Fikes, R., & Levesque, H. (1983). KRYPTON: A functional approach to knowledge representation. *Computer*, *16*, 67–73.

Carbonell, J. G., Blythe, J., Etzioni, O., Gil, Y., Joseph, R., Kahn, D., Knoblock, C., Minton, S., Pérez, A., Reilly, S., Veloso, M., & Wang, X. (1992). Prodigy 4.0: The manual and tutorial. Tech. rep. CMU-CS-92-150, School of Computer Science, Carnegie Mellon University.

Cresswell, S., & Coddington, A. M. (2004). Adapting LPGP to plan with deadlines. In de Mántaras, R. L., & Saitta, L. (Eds.), *Proceedings of the Sixteenth Eureopean Conference on Artificial Intelligence (ECAI-2004)*, pp. 983–984, Valencia, Spain. IOS Press, Amsterdam, The Netherlands.

Cresswell, S., Fox, M., & Long, D. (2002). Extending TIM domain analysis to handle ADL constructs. In McCluskey, T. L. (Ed.), *Proceedings of the AIPS-2002 Workshop on Knowledge Engineering Tools and Techniques for A.I. Planning*. Available at http://www.cis.strath.ac.uk/research/publications/papers/strath_cis_publication_80.pdf.

Currie, K., & Tate, A. (1991). O-Plan: The open planning architecture. *Artificial Intelligence*, *52*(1), 49–86.

de Kleer, J., & Brown, J. S. (1984). A qualitative physics based on confluences. *Artificial Intelligence*, *24*(1–3), 7–83.

Doherty, P. (1994). Reasoning about action and change using occlusion. In Cohn, A. G. (Ed.), *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI-1994)*, pp. 401–405, Amsterdam, The Netherlands. John Wiley and Sons, Chichester, England. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/ecai94.ps.gz.

Doherty, P. (1996). PMON$^+$: A fluent logic for action and change, formal specification, version 1.0. Tech. rep. LITH-IDA-96-33, Department of Computer and Information Science, Linköping University, Linköping, Sweden. Available at http://www.ida.liu.se/publications/techrep/96/tr96.html.

Doherty, P. (2004). Advanced research with autonomous unmanned aerial vehicles. In Dubois, D., Welty, C., & Williams, M.-A. (Eds.), *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR-2004)*, pp. 731–732, Whistler, British Columbia, Canada. AAAI Press, Menlo Park, California, USA. Extended abstract for plenary talk.

Doherty, P., Grabowski, M., Łukaszewicz, W., & Szałas, A. (2003). Towards a framework for approximate ontologies. *Fundamenta Informaticae*, *57*(2-4), 147–165.

Doherty, P., Granlund, G., Kuchcinski, K., Sandewall, E., Nordberg, K., Skarman, E., & Wiklund, J. (2000). The WITAS unmanned aerial vehicle project. In Horn, W. (Ed.), *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-2000)*, pp. 747–755, Berlin, Germany. IOS Press, Amsterdam, The Netherlands.

Doherty, P., & Gustafsson, J. (1998). Delayed effects of actions = direct effects + causal rules. *Linköping Electronic Articles in Computer and Information Science*, *3*. Available at http://www.ep.liu.se/ea/cis/1998/001.

Doherty, P., Gustafsson, J., Karlsson, L., & Kvarnström, J. (1998). TAL: Temporal Action Logics – language specification and tutorial. *Electronic Transactions on Artificial Intelligence*, *2*(3–4), 273–306. Available at http://www.ep.liu.se/ej/etai/1998/009/.

Doherty, P., Haslum, P., Heintz, F., Merz, T., Persson, T., & Wingman, B. (2004). A distributed architecture for intelligent unmanned aerial vehicle experimentation. In Alami, R. (Ed.), *Proceedings of the Seventh International Symposium on Distributed Autonomous Robotic Systems (DARS-2004)*, Toulouse, France. Springer-Verlag.

Doherty, P., Kachniarz, J., & Szałas, A. (2003). Using contextually closed queries for local closed-world reasoning in rough knowledge databases. In Pal, S., Polkowski, L., & Skowron, A. (Eds.), *Rough-Neuro Computing: Techniques for Computing with Words*, Cognitive Technologies, chap. 9, pp. 219–250. Springer-Verlag New York.

Doherty, P., & Kvarnström, J. (1998). Tackling the qualification problem using fluent dependency constraints: Preliminary report. In Khatib, L., & Morris, R. (Eds.), *Proceedings of the Fifth International Workshop on Temporal Representation and Reasoning (TIME-1998)*, pp. 97–104, Los Alamitos, California, USA. IEEE Computer Society Press.

Doherty, P., & Kvarnström, J. (1999). TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. In Dixon, C., & Fisher, M. (Eds.), *Proceedings of the Sixth International Workshop on Temporal Representation and Reasoning (TIME-1999)*, pp. 47–54, Orlando, Florida, USA. IEEE Computer Society Press. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/time99-final.ps.gz.

Doherty, P., & Kvarnström, J. (2001). TALplanner: A temporal logic-based planner. *AI Magazine*, *22*(3), 95–102. See also http://www.aaai.org/Library/Magazine/Vol22/22-03/vol22-03.html.

Doherty, P., & Łukaszewicz, W. (1994). Circumscribing Features and Fluents: A fluent logic for reasoning about action and change. In Gabbay, D. M., & Ohlbach, H. J. (Eds.), *Proceedings of the First International Conference on Temporal Logic (ICTL-1994)*, Vol. 827 of *Lecture Notes in Artificial Intelligence*, pp. 82–100. Springer Verlag London.

Doherty, P., Łukaszewicz, W., & Szałas, A. (2000). Efficient reasoning using the local closed-world assumption. In Cerri, S. A., & Dochev, D. (Eds.), *Proceedings of the Ninth International Conference on Artificial Intelligence: Methodology, Systems and Applications (AIMSA-2000)*, Vol. 1904 of *Lecture Notes in Artificial Intelligence*, pp. 49–58, Varna, Bulgaria. Springer-Verlag.

Doherty, P., Łukaszewicz, W., & Szałas, A. (2003a). Approximation transducers and trees: A technique for combining rough and crisp knowledge. In Pal, S., Polkowski, L., & Skowron, A. (Eds.), *Rough-Neuro Computing: Techniques for Computing with Words*, Cognitive Technologies, chap. 8, pp. 189–218. Springer-Verlag New York.

Doherty, P., Łukaszewicz, W., & Szałas, A. (2003b). Information granules for intelligent knowledge structures. In Wang, G., Liu, Q., Yao, Y., & Skowron, A. (Eds.), *Proceedings of the Ninth Internationall Conference on Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing (RSFDGrC-2003)*, Vol. 2639 of *Lecture Notes in Artificial Intelligence*, pp. 405–412, Chongqing, China. Springer-Verlag.

Doherty, P., Łukaszewicz, W., & Szałas, A. (2003c). Tolerance spaces and approximative representational structures. In Günter, A., Kruse, R., & Neumann, B. (Eds.), *Proceedings of the 26th Annual German Conference on Artificial Intelligence (KI-2003)*, Vol. 2821 of *Lecture Notes in Artificial Intelligence*, Hamburg, Germany. Springer.

Doherty, P., Łukaszewicz, W., & Szałas, A. (2004a). Approximative query techniques for agents using heterogeneous ontologies. In Dubois, D., Welty, C., & Williams, M.-A. (Eds.), *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR-2004)*, pp. 459–468, Whistler, British Columbia, Canada. AAAI Press, Menlo Park, California, USA.

Doherty, P., Łukaszewicz, W., & Szałas, A. (2004b). Approximative query techniques for agents with heterogeneous perceptual capabilities. In Svensson, P., & Schubert, J. (Eds.), *Proceedings of the Seventh International Conference on Information Fusion (Fusion-2004)*, pp. 175–182, Stockholm, Sweden. International Society of Information Fusion, Mountain View, California, USA.

Doherty, P., & Szałas, A. (2004). On the correspondence between approximations and similarity. In Tsumoto, S., Slowinski, R., Komorowski, J., & Grzymala-Busse, J. W. (Eds.), *Proceedings of the Fourth International Conference on Rough Sets and Current Trends in Computing (RSCTC-2004)*, Vol. 3066 of *Lecture Notes in Artificial Intelligence*, Uppsala, Sweden. Springer-Verlag.

Edelkamp, S., & Hoffmann, J. (2004). PDDL2.2: The language for the classical part of the fourth international planning competition. Tech. rep. 195, Albert Ludwigs Universität, Institut für Informatik, Freiburg, Germany. Available at http://www.mpi-sb.mpg.de/~hoffmann/publications.html.

Emerson, E. A. (1990). *Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics*, chap. Temporal and Modal Logic, pp. 997–1072. Elsevier and MIT Press.

Erol, K., Hendler, J. A., & Nau, D. S. (1994). UMCP: A sound and complete procedure for hierarchical task-network planning. In Hammond, K. J. (Ed.), *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-1994)*, pp. 249–254, Chicago, Illinois. AAAI Press, Menlo Park, California, USA.

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*(3–4), 189–208.

Finger, J. J. (1987). *Exploiting Constraints in Design Synthesis*. Ph.D. thesis, Stanford University, Stanford, California, USA.

Fourman, M. P. (2000). Propositional planning. In *Proceedings of the AIPS-2000 Workshop on Model-Theoretic Approaches to Planning*. Available at http://homepages.inf.ed.ac.uk/mfourman/tools/propplan/PlanningPaper/Planning.pdf.

Fox, M., & Long, D. (1998). The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, *9*, 367–421. Available at http://www.jair.org/contents/v9.html.

Fox, M., & Long, D. (1999). The detection and exploitation of symmetry in planning problems. In Dean, T. (Ed.), *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-1999)*, pp. 956–961, Stockholm, Sweden. Morgan Kaufmann Publishers, San Francisco, California, USA.

Fox, M., & Long, D. (2000a). Automatic synthesis and use of generic types in planning. In Chien, S., Kambhampati, S., & Knoblock, C. A. (Eds.), *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000)*, pp. 196–205, Breckenridge, Colorado, USA. AAAI Press, Menlo Park, USA.

Fox, M., & Long, D. (2000b). Utilizing automatically inferred invariants in graph construction and search. In Chien, S., Kambhampati, S., & Knoblock, C. A. (Eds.), *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000)*, pp. 102–111, Breckenridge, Colorado, USA. AAAI Press, Menlo Park, California, USA.

Fox, M., & Long, D. (2001a). PDDL+ level 5: An extension to PDDL2.1 for modelling planning domains with continuous time-dependent effects. Available at http://www.dur.ac.uk/d.p.long/pddllevel5.ps.gz.

Fox, M., & Long, D. (2001b). PDDL2.1: An extension to PDDL for expressing temporal planning domains. Available at http://www.dur.ac.uk/d.p.long/pddl2.ps.gz.

Fox, M., & Long, D. (2002). Extending the exploitation of symmetries in planning. In Ghallab, M., Hertzberg, J., & Traverso, P. (Eds.), *Proceedings of the Sixth Interna-*

tional Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002), pp. 83–91, Toulouse, France. AAAI Press, Menlo Park, California, USA.

Fox, M., & Long, D. (2003). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, *20*, 61–124. Available at http://www.jair.org/contents/v20.html.

Gerevini, A., & Schubert, L. K. (1998). Inferring state constraints for domain-independent planning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI-1998 / IAAI-1998)*, pp. 905–912, Madison, Wisconsin, USA. AAAI Press, Menlo Park, California, USA / The MIT Press, Cambridge, Massachusetts, USA.

Gerevini, A., & Schubert, L. K. (2000). Discovering state constraints in DISCOPLAN: Some new results. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence (AAAI-2000 / IAAI-2000)*, pp. 761–767, Austin, Texas, USA. AAAI Press, Menlo Park, California, USA / The MIT Press, Cambridge, Massachusetts, USA.

Ghallab, M., Howe, A. E., Knoblock, C., McDermott, D., Ram, A., Veloso, M. M., Weld, D. S., & Wilkins, D. (1998). PDDL—the planning domain definition language. Technical report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, Connecticut, USA.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, San Francisco, California, USA.

Ginsberg, M. L., & Smith, D. E. (1988). Reasoning about action II: The qualification problem. *Artificial Intelligence*, *35*(3), 311–342.

Giunchiglia, E., & Lifschitz, V. (1995). Dependent fluents. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-1995)*, pp. 1964–1969, Montréal, Québec, Canada. Morgan Kaufmann Publishers, San Mateo, California, USA.

Green, C. (1969). Applications of theorem proving to problem solving. In *Proceedings of the First International Joint Conference on Artificial Intelligence (IJCAI-1969)*. Morgan Kaufmann.

Gupta, N., & Nau, D. S. (1992). On the complexity of blocks-world planning. *Artificial Intelligence*, *56*(2-3), 223–254.

Gustafsson, J. (2001). *Extending Temporal Action Logic*. Ph.D. thesis, Linköping Studies in Science and Technology, Dissertation No. 689.

Gustafsson, J., & Doherty, P. (1996). Embracing occlusion in specifying the indirect effects of actions. In Aiello, L. C., Doyle, J., & Shapiro, S. C. (Eds.), *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR-1996)*, pp. 87–98. Morgan Kaufmann Publishers, San Francisco, California, USA. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/final-kr96.ps.gz.

Gustafsson, J., & Kvarnström, J. (2001). Elaboration tolerance through object-orientation. In *Proceedings of the Fifth Symposium on Logical Formalizations of Commonsense Reasoning (Common Sense-2001)*. Available at http://www.cs.nyu.edu/faculty/davise/commonsense01/final/kvarnstrom.ps.

Gustafsson, J., & Kvarnström, J. (2004). Elaboration tolerance through object-orientation. *Artificial Intelligence*, *153*, 239–285.

Hanks, S., & McDermott, D. V. (1986). Default reasoning, nonmonotonic logics, and the frame problem. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-1986)*, pp. 328–333, Philadelphia, Pennsylvania, USA. Morgan Kaufmann Publishers, Los Altos, California, USA.

Haslum, P., & Jonsson, P. (2000). Planning with reduced operator sets. In Chien, S., Kambhampati, S., & Knoblock, C. A. (Eds.), *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000)*, pp. 150–158, Breckenridge, Colorado, USA. AAAI Press, Menlo Park, California, USA.

Heintz, F. (2001). Chronicle recognition in the WITAS UAV project – a preliminary report.. Presented at the Swedish AI Society Workshop (SAIS), Skövde, Sweden.

Heintz, F., & Doherty, P. (2004a). DyKnow: A framework for processing dynamic knowledge and object structures in autonomous systems. In *International Workshop on Monitoring, Security, and Rescue Techniques in Multiagent Systems (MSRAS-2004)*, Plock, Poland. Springer-Verlag.

Heintz, F., & Doherty, P. (2004b). Dyknow: An approach to middleware for knowledge processing. *Journal of Intelligent and Fuzzy Systems*, *15*(1), 3–13.

Heintz, F., & Doherty, P. (2005). DyKnow: A framework for processing dynamic knowledge and object structures in autonomous systems. In Dunin-Keplicz, B., Jankowski, A., Skowron, A., & Szczuka, M. (Eds.), *Monitoring, Security, and Rescue Techniques in Multiagent Systems*, Advances in Soft Computing, pp. 479–492. Springer-Verlag Heidelberg.

Henschel, A., & Thielscher, M. (1999). The LMW traffic world in the fluent calculus.. Available at http://www.ida.liu.se/ext/etai/lmw/.

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, *14*, 253–302. Available at http://www.jair.org/contents/v14.html; planner available at http://www.mpi-sb.mpg.de/~hoffmann/ff.html.

Immerman, N. (1998). *Descriptive Complexity*. Texts in Computer Science. Springer-Verlag New York.

Karlsson, L., & Gustafsson, J. (1999). Reasoning about concurrent interaction. *Journal of Logic and Computation*, *9*(5), 623–650.

Karlsson, L., Gustafsson, J., & Doherty, P. (1998). Delayed effects of actions. In Prade, H. (Ed.), *Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI-1998)*, pp. 542–546, Brighton, UK. John Wiley and Sons, Chichester, England. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/ecai98.ps.gz.

Kautz, H., & Selman, B. (1998). BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working notes of the Workshop on Planning as Combinatorial Search (in conjunction with AIPS-1998)*, Pittsburgh, Pennsylvania, USA. AAAI Press, Menlo Park, California, USA. See http://www.cs.washington.edu/homes/kautz/satplan/blackbox/.

Kautz, H., & Selman, B. (1999). Unifying SAT-based and graph-based planning. In Dean, T. (Ed.), *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-1999)*, pp. 318–325, Stockholm, Sweden. Morgan Kaufmann Publishers, San Francisco, California, USA. See http://www.cs.washington.edu/homes/kautz/satplan/blackbox/.

Kibler, D., & Morris, P. (1981). Don't be stupid. In Drinan, A. (Ed.), *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-1981)*, pp. 345–347, Vancouver, British Columbia, Canada.

Koehler, J. (2000). Miconic 10 elevator domain web page.. http://www.informatik.uni-freiburg.de/~koehler/elev/elev.html.

Koehler, J., Nebel, B., Hoffmann, J., & Dimopoulos, Y. (1997). Extending planning graphs to an ADL subset. In Steel, S. (Ed.), *Proceedings of the Fourth European Conference on Planning (ECP-1997)*, Vol. 1348 of *Lecture Notes in Computer Science*, pp. 273–285, Toulouse, France. Springer-Verlag. Available at http://www.mpi-sb.mpg.de/~hoffmann/papers/ecp97.ps.gz.

Koehler, J. (2001). From theory to practice: AI planning for high performance elevator control. In F. Baader, G. Brewka, T. E. (Ed.), *Proceedings of the Joint German/Austrian Conference on AI: Advances in Artificial Intelligence (KI-2001)*, Vol. 2174 of *Lecture Notes in Computer Science*, pp. 459–462, Vienna, Austria. Springer-Verlag.

Koehler, J., & Ottiger, D. (2002). An AI-based approach to destination control in elevators. *AI Magazine*, *23*(3), 59–78. See also http://www.aaai.org/Library/Magazine/Vol23/23-03/vol23-03.html.

Koehler, J., & Schuster, K. (2000). Elevator control as a planning problem. In Chien, S., Kambhampati, S., & Knoblock, C. A. (Eds.), *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000)*, pp. 331–338, Breckenridge, Colorado, USA. AAAI Press, Menlo Park, California, USA.

Koubarakis, M. (1994). Complexity results for first-order theories of temporal constraints. In Doyle, J., Sandewall, E., & Torasso, P. (Eds.), *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-1994)*, pp. 379–390. Morgan Kaufmann Publishers, San Francisco, California, USA.

Kvarnström, J. (1997–2005). VITAL. An on-line system for reasoning about action and change using TAL. Software available at http://www.ida.liu.se/~jonkv/vital/.

Kvarnström, J. (2002). Applying domain analysis techniques for domain-dependent control in TALplanner. In Ghallab, M., Hertzberg, J., & Traverso, P. (Eds.), *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, pp. 101–110, Toulouse, France. AAAI Press, Menlo Park, California, USA.

Kvarnström, J., & Doherty, P. (2000a). Tackling the qualification problem using fluent dependency constraints. *Computational Intelligence*, *16*(2), 169–209.

Kvarnström, J., & Doherty, P. (2000b). TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, *30*, 119–169.

Kvarnström, J., Doherty, P., & Haslum, P. (2000). Extending TALplanner with concurrency and resources. In Horn, W. (Ed.), *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-2000)*, Frontiers in Artificial Intelligence and Applications, pp. 501–505, Berlin, Germany. IOS Press, Amsterdam, The Netherlands. Available at ftp://ftp.ida.liu.se/pub/labs/kplab/people/patdo/www-ecai.ps.gz.

Kvarnström, J., & Magnusson, M. (2003). TALplanner in the Third International Planning Competition: Extensions and control rules. *Journal of Artificial Intelligence Research*, *20*, 343–377. Available at http://www.jair.org/contents/v20.html.

Lifschitz, V. (1987). Formal theories of action. In Brown, F. M. (Ed.), *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*, pp. 35–58, Lawrence, Kansas, USA. Morgan Kaufmann Publishers, Los Altos, California, USA.

Lifschitz, V. (2000). Missionaries and cannibals in the causal calculator. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR-2000)*, pp. 85–96. Morgan Kaufmann Publishers, San Francisco, California, USA.

Lin, F. (2001). A planner called R. *AI Magazine*, *22*(3), 73–76. See also http://www.aaai.org/Library/Magazine/Vol22/22-03/vol22-03.html.

Lin, F., & Reiter, R. (1994). State constraints revisited. *Journal of Logic and Computation*, *4*(5), 655–678.

Long, D., & Fox, M. (1999). Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research*, *10*, 87–115. Available at http://www.jair.org/contents/v10.html.

Long, D., & Fox, M. (2003). The third international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, *20*, 1–59. Available at http://www.jair.org/contents/v20.html.

McCain, N., & the Texas Action Group (1997). The causal calculator.. Available at http://www.cs.utexas.edu/users/tag/cc/.

McCain, N., & Turner, H. (1995). A causal theory of ramifications and qualifications. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-1995)*, Montréal, Québec, Canada. Morgan Kaufmann Publishers, San Francisco, California, USA.

McCarthy, J. (1959). Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pp. 75–91, London. Her Majesty's Stationary Office. Available at http://www-formal.stanford.edu/jmc/mcc59.html.

McCarthy, J. (1980). Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence*, *13*(1–2), 27–39. Reprinted in McCarthy (1990).

McCarthy, J. (1986). Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, *28*(1), 89–116. Reprinted in (McCarthy, 1990).

McCarthy, J. (1990). *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz*. Ablex.

McCarthy, J. (1998). Elaboration tolerance. In *The 1998 Symposium on Logical Formalizations of Commonsense Reasoning (Common Sense-1998)*, London. Available at http://www-formal.stanford.edu/jmc/elaboration.html.

McCarthy, J., & Hayes, P. (1969). Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, *4*, 463–502.

McDermott, D. (1998). AIPS98 planning competition results. http://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html.

Merz, T. (2004). Building a system for autonomous aerial robotics research. In *IFAC Symposium on Intelligent Autonomous Vehicles (IAV-2004)*, Lisbon, Portugal. Elsevier.

Morgenstern, L. (1998). Inheritance comes of age: Applying nonmonotonic techniques to problems in industry. *Artificial Intelligence*, *103*(1–2), 237–271.

Moss, C. (1994). *Prolog++, The power of object-oriented and logic programming*. Addison-Wesley.

Nau, D. S., Au, T. C., Ilghami, O., Kuter, U., Murdock, J. W., Wo, D., & Yaman, F. (2003). SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, *20*, 379–404. Available at http://www.jair.org/contents/v20.html.

Nau, D. S., Cao, Y., Lotem, A., & Muños-Avila, H. (2001). The SHOP planning system. *AI Magazine*, *22*(3), 91–94. See also http://www.aaai.org/Library/Magazine/Vol22/22-03/vol22-03.html.

Nau, D. S., Cau, Y., Lotem, A., & Muños-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. In Dean, T. (Ed.), *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-1999)*, pp. 968–973, Stockholm, Sweden. Morgan Kaufmann Publishers, San Francisco, California, USA. Available at http://www.cs.umd.edu/~nau/papers/shop-ijcai99.pdf.

Nebel, B., Dimopoulos, Y., & Koehler, J. (1997). Ignoring irrelevant facts and operators in plan generation. In *Proceedings of the Fourth European Conference on Planning (ECP-1997)*, pp. 338–350, Toulouse, France.

Pednault, E. P. D. (1989). ADL: Exploring the middle ground between STRIPS and the Situation Calculus. In Brachman, R. J., Levesque, H. J., & Reiter, R. (Eds.), *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR-1989)*, pp. 324–332, Toronto, Ontario, Canada. Morgan Kaufmann Publishers, San Mateo, California, USA.

Pettersson, P. O. (2003). Helicopter path planning using probabilistic roadmaps. Master's thesis, Linköping University. Available at http://www.ida.liu.se/~peope/.

Pettersson, P. O., & Doherty, P. (2004). Probabilistic roadmap based path planning for an autonomous unmanned aerial vehicle. In *Proceedings of the ICAPS-2004 Workshop on Connecting Planning Theory with Practice*. Fourteenth International Conference on Automated Planning and Scheduling, ICAPS'2004. Available at http://www.ida.liu.se/~peope/.

Rintanen, J. (2000a). Incorporation of temporal logic control into plan operators. In *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI-2000)*, pp. 526–530, Berlin, Germany. IOS Press, Amsterdam, The Netherlands.

Rintanen, J. (2000b). An iterative algorithm for synthesizing invariants. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence (AAAI-2000 / IAAI-2000)*, pp. 806–811, Austin, Texas, USA. AAAI Press, Menlo Park, California, USA / The MIT Press, Cambridge, Massachusetts, USA. Available at http://www.informatik.uni-freiburg.de/~rintanen/CV.html.

Sacerdoti, E. D. (1975). The nonlinear nature of plans. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI-1975)*, pp. 206–214, Tiblisi, Georgia, USSR.

Sandewall, E. (1994). *Features and Fluents: A Systematic Approach to the Representation of Knowledge about Dynamical Systems*, Vol. 1. Oxford University Press.

Sandewall, E. (1999). Logic modelling workshop: Communicating axiomatizations of actions and change. Available at http://www.ida.liu.se/ext/etai/lmw.

Sandewall, E., & Rönnquist, R. (1986). A representation of action structures. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-1986)*, pp. 89–97, Philadelphia, Pennsylvania, USA. Morgan Kaufmann Publishers, Los Altos, California, USA.

Scholz, U. (2000). Extracting state constraints from PDDL-like planning domains. In *Proceedings of the AIPS-2000 Workshop on Analyzing and Exploiting Domain Knowledge for Efficient Planning*, pp. 43–48.

Shanahan, M. (1997). *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. The MIT Press, Cambridge, Massachusetts, USA.

Shoham, Y. (1987). Nonmonotonic logics: Meaning and utility. In McDermott, J. P. (Ed.), *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-1987)*, pp. 388–393, Milan, Italy. Morgan Kaufmann Publishers, Los Altos, California, USA.

Slaney, J., & Thiébaux, S. (2001). Blocks world revisited. *Artificial Intelligence*, *125*(1-2), 119–153.

Smith, D. E. (2003). The case for durative actions: A commentary on PDDL2.1. *Journal of Artificial Intelligence Research*, *20*, 149–154. Available at http://www.jair.org/contents/v20.html.

Smith, D. E., & Weld, D. S. (1999). Temporal planning with mutual exclusion reasoning. In Dean, T. (Ed.), *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-1999)*, pp. 326–337, Stockholm, Sweden. Morgan Kaufmann Publishers, San Francisco, California, USA. Available at ftp://ftp.cs.washington.edu/pub/ai/ijcai99-tgp.ps.

Störr, H.-P. (2001). Planning in the fluent calculus using binary decision diagrams. *AI Magazine*, *22*(3), 103–106. See also http://www.aaai.org/Library/Magazine/Vol22/22-03/vol22-03.html and the BDDPlan web page at http://www.stoerr.net/bddplan.html.

Tate, A. (1977). Generating project networks. In Reddy, R. (Ed.), *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-1977)*, pp. 888–893, Cambridge, Massachusetts, USA.

Thielscher, M. (1996a). Causality and the qualification problem. In Aiello, L. C., Doyle, J., & Shapiro, S. C. (Eds.), *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR-1996)*, pp. 51–62, Cambridge, Massachusetts, USA. Morgan Kaufmann Publishers, San Francisco, California, USA.

Thielscher, M. (1996b). Qualification and causality. Tech. rep. TR-96-026, International Computer Science Institute (ICSI), Berkeley, California, USA.

Thielscher, M. (1997). Qualified ramifications. In Kuipers, B., & Webber, B. (Eds.), *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-*

1997), Providence, Rhode Island, USA. The MIT Press, Cambridge, Massachusetts, USA.

Thielscher, M. (1998). Introduction to the fluent calculus. *Electronic Transactions on Artificial Intelligence*, *2*(3–4), 179–192. Available at http://www.ep.liu.se/ej/etai/1998/006/.

Veloso, M. M., Carbonell, J. G., Pérez, A., Borrajo, D., Fink, E., & Blythe, J. (1995). Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, *7*(1), 81–120.

Vere, S. A. (1983). Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, *5*(3), 246–267.

Weld, D. S., Anderson, C. R., & Smith, D. E. (1998). Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI-1998 / IAAI-1998)*, pp. 897–904, Madison, Wisconsin, USA. AAAI Press, Menlo Park, California, USA / The MIT Press, Cambridge, Massachusetts, USA.

Wilkins, D. E. (1988). Causal reasoning in planning. *Computational Intelligence*, *4*(4), 373–380.

Winograd, T. (1972). *Understanding Natural Language*. Academic Press.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani:** Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming : A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund:** Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld:** Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén:** Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi:** Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-supported Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.